Universitatea „Politehnica” din București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Reprezentarea ternară a numerelor binare cu aplicaţie în înmulţirea rapidă

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de

Inginer în domeniul *Electronică și Telecomunicații*

programul de studii de licență *Electronică Aplicată*

Conducători științifici

Prof. Dr. Ing. Coreneliu BURILEANU

Ing. Radu Ion POP

Absolvent

Maria Cristina TEODORESCU

2015

Anexa 1

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul Electronică Aplicată şi Ingineria Informaţiei

Aprobat Director de Departament :
Prof. Dr. Ing. Sever Paşca

TEMA PROIECTULUI DE DIPLOMĂ
a studentului Teodorescu O.M. Maria Cristina, grupa 441 B

1. Titlul temei: Reprezentarea ternară a numerelor binare cu aplicaţie în înmulţirea rapidă

2. Contribuţia practică, originală a studentului va consta în:

- *Se va realiza un bloc de înmulţire rapidă (într-un singur tact) şi se va implementea pe un FPGA Virtex-7; operanzii se transferă printr-un port USB de la calculator, iar rezultatul este evidenţiat pe afişajul disponibil pe placă*
- *Se vor gasi ecuaţiile logice pentru adunare fără propagare de carry (compresoare 4:2)*
- *Multiplicarea se va face pe baza algoritmului Booth Radix-8 cu rezultat în Redundant Binary*
- *Implementarea hardware a blocului digital de înmulţire se va realiza utilizând Arborele Wallace calibrat pentru număr de biţi exprimat prin puteri ale lui 2*

3. Proiectul se bazează pe cunoştinţe dobîndite în principal la urmatoarele 3-4 discipline: Microcontrolere, Arhitectura Microprocesoarelor, Circuite Integrate Digitale

4. Proprietatea intelectuală asupra proiectului aparţine: *studentului*

5. Locul de desfăşurare a activităţii: *UPB - Sala nr. 3 de Calculatoare/Infineon Technologies*

6. Realizarea practică/ proiectul ramân în proprietatea: *studentului*

7. Data eliberării temei: 02.10.2014

CONDUCATORI LUCRARE:

Prof. Corneliu Burileanu

Ing. Radu Ion Pop

STUDENT:

Maria Cristina Teodorescu

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul " *Reprezentarea ternară a numerelor binare cu aplicație în înmulțirea rapidă*", prezentată în cadrul Facultății de Electronică, Telecomunicațiișsi Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații*, programul de studii *Electronică Aplicată* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, *01.07.2015*

Absolvent *Maria Cristina TEODORESCU*

Cuprinsul Lucrării

Lista figurilor

Lista tabelelor

Lista acronimelor

A

ACK = Acknowledge
ASCII = American Standard Code for Informational Interchange
ASIC = Application Specific Integrated Circuit

C

CGRAM = Character Generator Random Access Memory
CGROM = Character Generator Read Only Memory
COM = Communication Port
CTS = Clear to Send

D

DCE = Data Communications Equipment
DDRAM = Double Data Random Access Memory
DSP = Digital Signal Processor
DTE = Data Terminal Equipment

E

ETX = End of text

F

FIFO = First In First Out
FPGA = Field Programmable Gate Array

G

GPIO = General-purpose input/output

H

HEX = Hexadecimal

I

IDE = Integrated Design Environment
IO = Input/Output
ISE = Integrated Synthesis Environment

J

JTAG = Joint Test Action Group

L

LCD = Liquid Crystal Display
LVDS = Low-voltage differential signaling
LSB = Least Significant Bit

M

MSB = Most Significant Bit

P

PN = Positive - Negative
PNC = Positive – Negative – Complement

R

RB – Redundant Binary
RNS = Residue Number System
RTL = Register Transfer Level
RTS = Request to Send

S

SB = Sign Bit

U

UART = Universal Asynchronous receiver/transmitter
USB = Universal Serial Bus

V

VHDL = VHSIC Hardware Description Language
VHSIC = Very High Speed Integrated Circuit

INTRODUCERE

Prezenta lucrare își propune analiza etapelor de proiectare și a performanțelor unui bloc de înmulțire rapidă, capabil să livreze rezultatul după un ciclu de ceas de la primirea operanzilor. Mai mult decât atât, blocul de înmulțire utilizează reprezentarea ternară a numerelor binare, aceasta având următoarele avantaje față de reprezentarea utilizând aritmetica convențională: necesitatea unui număr mai mic de blocuri de însumare, reducând astfel înălțimea arborelui Wallace, și adunarea fără propagare de transport, înlocuind astfel adunarea numerelor în complement față de 2.

Obiectivele acestei lucrări, împreună cu afirmația anterioară, sunt după cum urmează:

1) proiectarea unui bloc de înmulțire ce utilizează aritmetică convențională
2) proiectarea unui bloc de înmulțire ce utilizează aritmetică redundantă
3) analiza performanțelor celor două blocuri de înmulțire
4) implementarea practică a blocului de înmulțire ce utilizează aritmetică redundantă utilizând o placă de dezvoltare de la firma Xilinx, cu cip FPGA Virtex- 7

Prezenta lucrare este organizată în patru capitole. Astfel, în prima parte a lucrării se vor explica noțiunile de artimetică convențînală și aritmetică neconvențională și diferențele dintre acestea. De asemenea, este prezentată o scurtă istorie a evoluției algoritmilor de înmulțire, precum și câteva criterii de care s-a ținut cont în vederea alegerii optime a resurselor hardware și software.

Cea de-a doua parte a lucrării motivează alegerea plăcii de dezvoltare VC707 cu FPGA Virtex-7. În cel de-al doilea capitol este realizată o comparație între diverse familii de FPGA-uri ale firmei Xilinx, apoi sunt prezentate în detaliu aspecte tehnice de interes ale plăcii de dezvoltare.

În cea de-a treia parte a lucrării se vor prezenta în detaliu noțiunile teoretice ce stau la baza algoritmilor utilizați în vederea proiectării blocului de înmulțire și a înțelegerii modului de funcționare al acestora. De asemenea, în cadrul acestui capitol sunt prezentate cele două blocuri de înmulțire, cel cu aritmetică convențională și cel cu aritmetică de performanță. Se vor detalia schemele bloc, precum și modul de funcționare și importanța fiecarui modul implementat.

În cel de-al patrulea capitol al lucrării se va prezenta proiectarea și implementarea blocului de înmulțire rapidă, plecând de la schema bloc, unde se va explica modul de funcționare al fiecărei componente principale în parte. În continuare se va prezenta protocolul de comunicație între calculator și FPGA, prin intermediului punții USB – to – UART și protocolul de comunicație între FPGA și afișajul LCD. În final, se vor evidenția rezultatele simulării întregului sistem și a modulelor componente.

1 NOȚIUNI TEORETICE, GENERALITĂȚI

1.1 ARITMETICA – PARTE A ÎNTREGULUI SISTEM

Aritmetica este una din cele mai vechi ramuri ale matematicii, fiind o ramură de bază a acesteia și constă în studiul numerelor, în special a proprietăților operațiilor dintre ele, precum: adunarea, scăderea, înmulțirea, împărțirea. Aritmetica este o parte elementară a teoriei numerelor.

Aritmetica convențională a fost aplicată cu succes pentru a implementa și studia numeroase probleme de matematică, fizică sau informatică. Majoritatea aplicațiilor existente au la bază suportul unei aritmetici bine puse la punct, iar acest lucru este evidențiat prin performanța acestora. Procesarea grafică și video 3D, tranzacțiile financiare sau jocurile video sunt doar câteva exemple de astfel de aplicații. În acest gen de aplicații, problemele de procesare a datelor, impuse de acestea, modifică cerințele de performanță.

Alegerea tipului de aritmetică folosită în realizarea unei aplicații este decisivă atât din punct de vedere hardware, cât și din punct de vedere software, întrucât influențează în mod direct atât performanța algoritmului propriu-zis, cât și performanța sistemului, aria de siliciu ocupată, consumul de putere sau viteza. Ținând cont de standardul de performanță impus și de numărul mare de operații realizate, marea majoritate a microprocesoarelor și a procesoarelor digitale de semnal (DSP) de ultimă generație realizează toate cele patru operații aritmetice: adunarea, scăderea, înmulțirea și împărțirea.

Studiind cu atenție arhitectura și implementarea procesoarelor, ajungem la concluzia că din cele patru operații aritmetice, adunarea este cea mai importantă. Rezultatul scăderii este echivalentul rezultatului adunării operanzilor negați. De asemenea, rezultatul înmulțirii sau împărțirii se poate obține în urma unor adunări sau scăderi succesive. În concluzie, adunarea este operația fundamentală și crucială în sistemele digitale.

Implementarea cu succes a unei aritmetici algebrice computaționale, ce îndeplinește în proporție de 100% cerințele aplicației propuse, are la baza diagrama din Fig. 1.1. În aceasta diagramă se ilustrează cei șase pași esențiali de urmat în timpul implementării.

Implementare

Când vine vorba de implementare, un pas esențial este alegerea eficientă a resurselor hardware și software astfel încât produsul final să fie cât mai eficient. Partea hardware, raportată la cerințele impuse de aplicație, reprezintă o alegere dintre un layout realizat specific pentru aplicația curentă, folosirea unui procesor (ARM, procesor digital de semnal etc.), folosirea unui FPGA (Field Programmable Gate Array), folosirea unui ASIC (circuit integrat specific asincron) etc. Eficiența circuitului va depinde de ce resursă hardware se va folosi pentru implementare. Pe partea software, algoritmii sunt disponibili sub formă de librării predefinite. Aici, provocarea este să se folosească primitivele aritmetice în algoritmul dezvoltat în vedere eficientizării și sintetizării cât mai eficiente, raportat la cerințele aplicației dezvoltate.

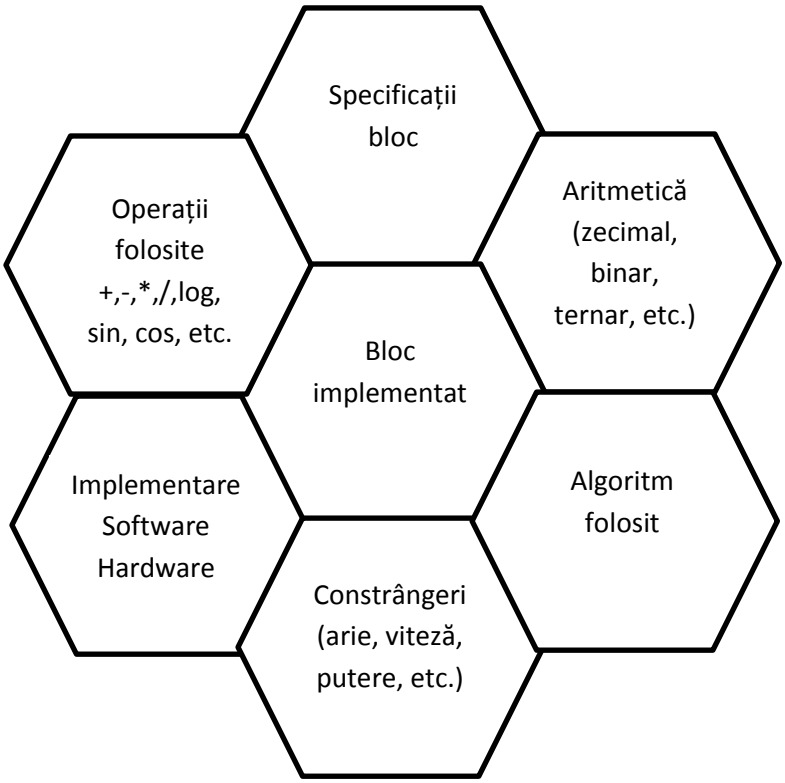

Fig. 1.1 Diagrama de realizare a unui bloc propus

Operații folosite

Alegerea operațiilor necesare este bazată pe cerințele aplicației realizate. Astfel, se va ține cont de scopul aplicației și se va încerca alegerea optimă pentru îndeplinirea acestuia. Se vor stabili operațiile necesare ce trebuie aplicate asupra operanzilor, în vederea obținerii rezultaului dorit, apoi se vor stabili operațiile intermediare în urma cărora se obține acelaşi rezultat. De exemplu, rezultatul înmulțirii se poate obține prin mai multe operții de adunare succesive aplicate operanzilor.

Aritmetică

Sistemele digitale sunt direct influențate de tipul de aritmetică folosit și de asemenea, de tipul de reprezentare. Alegerea unei aritmetici convenționale sau neconvenționale, mod de lucru în virgulă fixă sau mobilă, sau reprezentarea binară sau ternară se va face comform cu scopul aplicației și de constrângerile de implementare. În cazul unor implementari harware a unor aplicații ce necesită răspuns rapid, au consum mare de putere sau necesită un timp îndelungat de procesare a datelor primite, alegerea unei aritmetici convenționale nu este neaparat cea mai bună alegere. În acest caz, se trece la ideea folosirii unei aritmetici neconvenționale, precum reprezentarea ternară a numerelor binare cu semn, numită și aritmetică redundantă.

Algoritm folosit

Înainte de începerea implementării propriu-zise, se va încerca găsirea unui algoritm potrivit pentru a satisface cerințele aplicației și, de asemenea, pentru a folosi eficient resursele hardware disponibile. În acest pas trebuie avut în vedere că, deși un algoritm poate părea optim, nu este obligatoriu să fie comptibil cu implementarea software sau hardware, ducând astfel la performanțe slabe ale întregului sistem.

Specificațiile și constrângerile blocului implementat

Specificațiile blocului implementat sunt îndeplinite de operațiile folosite, tipul de aritmetică si algoritmul ales, în timp ce constrangerile sunt date de limitările acestui bloc, limitări ce trebuie respectate de aceste specificații.

1.2 SISTEME DE NUMERAȚIE CONVENȚIONALĂ

Setul de instrucțiuni specific unui procesor digital de semnal sau al unui procesor de uz general implică cel puțin un tip de adunare. De asemenea, operații aritmetice precum scăderea, înmulțirea sau împărțirea presupun efectuarea cel puțin unei adunări în vederea obținerii rezultatului final. De cele mai multe ori, prezența unui sumator sau a unui bloc ce conține mai multe sumatoare este obligatorie, de aceea performanța întregului sistem va fi determinată de performanța tipului de sumator ales. Astfel, este benefic pentru întregul sistem alegerea cu grijă a tipului de sumator implementat, întrucât poate afecta funcționarea optima a întregului circuit.

Reprezentarea și gestionarea numerelor eficient este un criteriu important ce trebuie avut în vedere în orice tip de aplicație, indiferent de domeniul de utilizare al acesteia, fie că e vorba de știință, inginerie sau finanțe. În viața de zi cu zi, cel mai răspândit sistem de reprezentare este reprezentarea zecimală. Orice numar întreg, de exemplu – X, poate fi reprezentat astfel:

$$X = x_{n-1} \cdot r^{n-1} + \cdots + x_1 \cdot r^1 + x_0 = \sum_{i=0}^{n-1} x_i \cdot r^i \quad (1.1)$$

În ecuația de mai sus, r reprezinta rădăcina sistemului, în cazul reprezentării zecimale fiind egal cu 10. Cifrele x_i sunt numere intregi din intervalul {0,1,…,r-1}, în cazul sistemului zecimal fiind {0,1,…,9}. Astfel, numărul întreg 25 se poate reprezenta în system de reprezentare zecimal prin:

$$25 = 2 \cdot 10^1 + 5$$

Având în vedere că în acest sistem fiecare număr întreg are o reprezentare unică, putem numi reprezentarea zecimală ca fiind o reprezentare convențională sau nonredundantă. O repreentare nonredundantă presupune că în cadrul unui sistem, o combinație de cifre x_i corespunde unui singur număr. Cu alte cuvinte, nu există două numere carora să le corespună aceeași combinație de cifre x_i.

Dacă reprezentarea zecimală a numerelor era ce mai întâlnit tip de reprezentare în viața de zi cu zi, în lumea digitală se folosește cu precădere reprezentarea binară a numerelor.

Ecuația de mai sus se poate aplica pentru orice tip de reprezentare. Astfel, în cazul reprezentarii numerelor binare, fără semn, vom avea urmatoarea ecuație:

$$X = x_{n-1} \cdot 2^{n-1} + \cdots + x_1 \cdot 2^1 + x_0 = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad (1.2)$$

În acest caz, cifrele x_i sunt numere întregi din intervalul {0,1}. De asemenea, și acest tip de reprezentare este una nonredundantă, deoarece fiecărui număr îi corespunde o combinatie unică de cifre x_i.

În cazul reprezentării binare a numerelor cu semn avem la dispoziție trei tipuri de reprezentare:

- Reprezentare în semn și magnitudine
- Reprezentare în complement față de 1
- Reprezentare în complement față de 2

Având în vedere ca reprezentarea în complement față de 2 este cea mai des folosită în lumea digital ne vom îndrepta interesul către aceasta. Astfel, ecuația pe baza căreia se transformă un număr zecimal este urmatoarea:

$$X = sgn \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \cdots + x_1 \cdot 2^1 + x_0 = sgn \cdot 2^n + \sum_{i=0}^{n-1} x_i \cdot 2^i \quad (1.3)$$

Variabila sgn reprezintă semnul numărului din zecimal și poate lua valorile 0 sau -1, unde 0 – reprezintă semul unui număr pozitiv și -1 – reprezintă semnul unui număr negativ.

Există numeroase avantajele ce rezultă în urma reprezentării numerelor în binar, precum:

- ușurința comparării a două numere prin verificarea biților cei mai semnificativi
- intervalul de reprezentare poate fi extins cu ușurință, doar prin adaugarea de cifre – adăugarea unei cifre are ca efect matematic înmulțirea cu 2 a întregului număr
- existența unui transport poate fi detectată cu ușurința doar prin verificarea bitului cel mai semnificativ
- semnul numărului poate fi dedus prin verificarea celui mai semnificativ bit

1.3 SUMATOR RIPPLE-CARRY

Operația de adunare bit cu bit a doi operanzi se poate realiza folosind mai multe sumatoare complete. Un circuit sumator complet are trei intrări si două ieșiri. Operațiile realizate de un circuit sumator complet sunt descrise de ecuațiile prezentate în continuare.

$$S_i = X_i \oplus Y_i \oplus C_{in} \quad (1.4)$$

$$C_{out} = (X_i \wedge Y_i) \vee \left(C_{in} \wedge (X_i \oplus Y_i)\right) = Majority\,(X_i, Y_i, C_{in}) \quad (1.5)$$

Următoarele notații se vor folosi în continuare pentru operațiile logice, în vederea evitării ambiguității.

$$x \vee y \leftrightarrow x\ OR\ y \quad (1.6)$$

$$x \wedge y \leftrightarrow x\ AND\ y \quad (1.7)$$

$$x \oplus y \leftrightarrow x\ XOR\ y \quad (1.8)$$

$$\bar{x} \leftrightarrow NOT\ x \quad (1.9)$$

X_i	Y_i	C_{in}	S_i	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabel 1.1 Tabelul de adevar pentru un sumator complet [1]

Din ecuațiile de mai sus reiese că operația de adunare se realizează prin intermediul a două porți XOR, iar transportul este obținut prin intermediul a două porți AND și o poartă OR. Cu toate acestea, această reprezentare nu este unică, expresia logică determinând structura circuitului. Din aceste considerente, există numeroase tipuri de sumatore, lăsând la latitudinea urilizatorului sa aleaga ce tip de sumator complet se potrivește sistemului implementat.

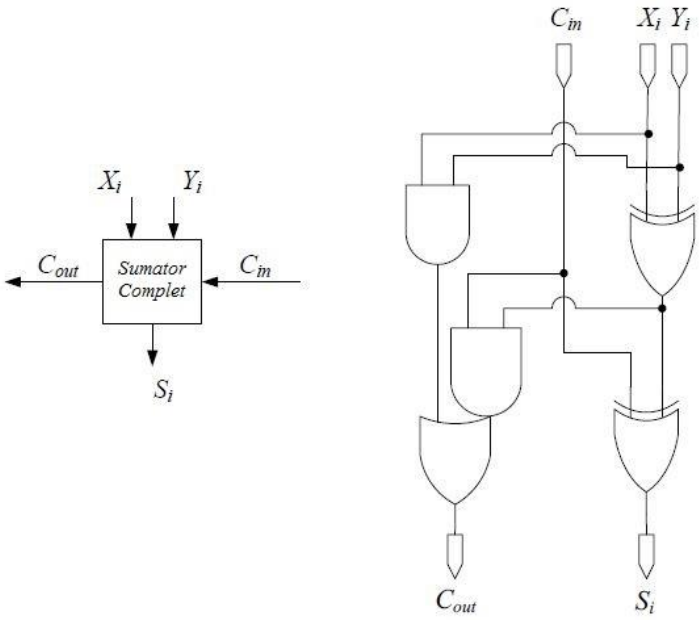

Fig. 1.2 Diagrama bloc și circuitul pentru un sumator complet [1]

În figura următoare este prezentată structura unui sumator Ripple-Carry. Acest sumator poate însuma operanzi cu n biți, iar în funcție de „n” este determinat numărul de sumatoare complete existente în sistem. Rezultatul este obținut prin însumarea bit cu bit a operanzilor, astfel, pentru pozția „i” este suma valorilor biților de pe poziția „i” a operanzilor plus transportul anterior, de pe poziția „i-1”.

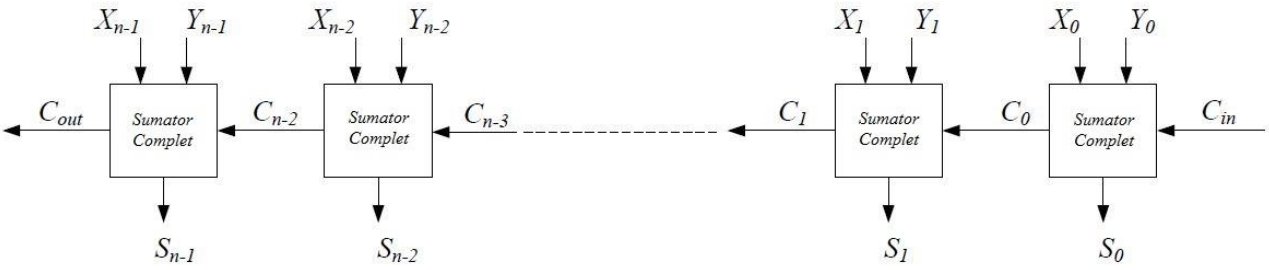

Fig. 1.3 Diagrama bloc a unui sumator Ripple-Carry [1]

Principalul avantaj al acestui tip de sumator este ușurința de realizare si ocuparea eficientă a ariei. Dezanatajul său este întârzierea propagată dintr-o celulă în alta, pana la bitul cel mai semnificativ (MSB), întârzierea fiind estimanată la 2n.

1.4 SUMATOR PARALEL CU PREFIX

Operația de adunare poate fi interpretată și ca o operatie cu prefix, în sensul că fiecare ieșire S_i depinde de toți biții de intrare de la 1 la i. În loc să se ia în calcul transportul de la un bloc la altul, acesta este extins la un grup de o lungime arbitrară. Astfel, în cadrul unui bloc un transport poate fi generat, propagat sau absorbit de blocul respectiv. În cazul în care transportul este propagat, acesta va aparea ca ieșire a blocului. În consecință, sumatoarele paralele cu prefix prezintă semnale generate g_i și semnale propagate p_i pentru a calcula transportul de la bitul cel mai puțin semnificativ (LSB) la bitul cel mai mult semnificativ (MSB).

Expresiile care descriu sumatorul paralel cu prefix sunt următoarele:

$$g_i = \begin{cases} (x_0 \wedge y_0) \vee \left(c_0 \wedge (x_0 \vee y_0)\right) & , i = 0 \\ x_i \wedge y_i & , i \neq 0 \end{cases} \quad (1.10)$$

$$p_i = x_i \oplus y_i \quad (1.11)$$

$$\left(G_{i:I}^0, P_{i:i}^0\right) = (g_i, p_i) \quad (1.12)$$

$$\begin{aligned} \left(G_{i:k}^l, P_{i:k}^l\right) &= \left(G_{i:j}^{l-1}, P_{i:j}^{l-1}\right) \cdot \left(G_{j:k}^{l-1}, P_{j:k}^{l-1}\right) \\ &= \left(G_{i:j}^{l-1} \vee \left(P_{i:j}^{l-1} \wedge G_{j:k}^{l-1}\right)\right), \left(P_{i:j}^{l-1} \wedge G_{j:k}^{l-1}\right) \end{aligned} \quad (1.13)$$

$$c_{i+1} = G_{i:0}^{m} \ (1.14)$$

$$s_i = p_i \oplus c_i \ (1.15)$$

Pentru $i = 0,1,\dots,n-1$ și $l = 1,\dots,m$, unde x_i și y_i sunt intrări, semnalele $G_{i:k}^{l}$ și $P_{i:k}^{l}$ reprezintă semnalele generate și propagate pentru grupul de biți $i, i-1,\dots,k$ cu prefixul de nivel l. În funcție de modul de evaluare al operatorului asociativ cu prefix $\cdot$, vor exista diferite arhitecturi ale arborelui de adunare.

Structura generala a unui sumator paralel cu prefix este evidențiata în figura 1.4 și conține trei componente majore: componenta de pre-procesare, arborele cu prefix ți componenta de post-procesare.

Există două posibilități de procesare a transportului de intrare: abordare înceată figura 1.4 sau abordare rapidă figura 1.5. În analiza performanței algoritmului este analizat doar arborele cu prefix, în timp ce componentele de pre-procesare și post-procesare sunt considerate echivalente pentru toate arhitecturile de sumator. De asemenea, în unele cazuri, nu este respectata întru totul schema generală, unele arhitecturi neavând nevoie de componenta de pre-procesare.

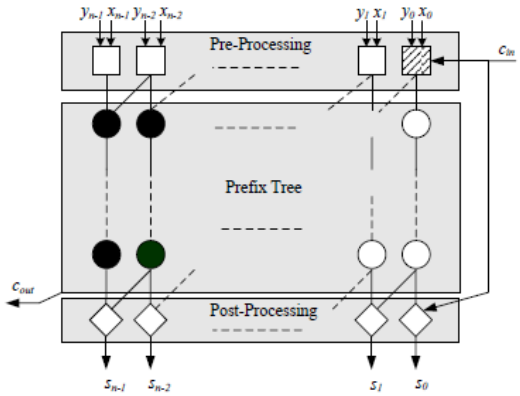

Fig. 1.4 Structura generală a unui sumator paralel cu prefix cu procesare înceată a transportului de intrare [1]

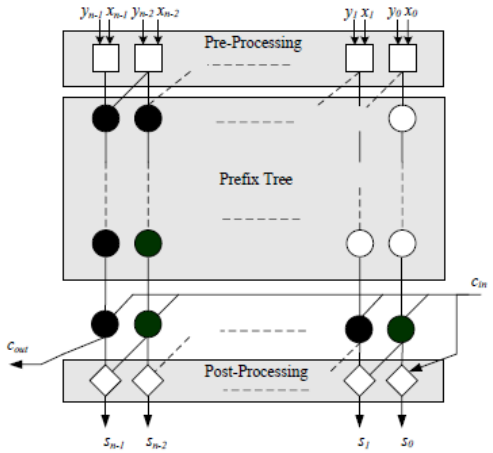

Fig. 1.5 Structura generală a unui sumator paralel cu prefix cu procesare rapidă a transportului de intrare [1]

O metodă des întâlnită în analiza algoritmilor este vizualizarea sumatoarelor cu prefix folosind o diagrama cu noduri. În arborele cu prefix sunt folosite două tipuri de noduri: noduri negre și noduri albe. Nodurile negre acceptă două perechi de biți generați și propagați și produc un grup de semnale de propagare și generare. Nodurile albe din diagramă nu implementează o funcție logică, scopul lor principal fiind copierea intrării la ieșire.

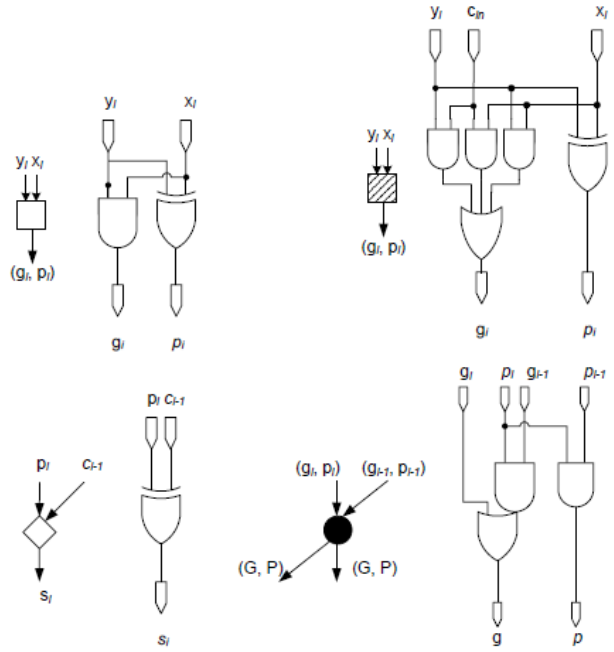

Fig. 1.6 Implementarea unui nod în interiorul unui arbore cu prefix paralel [1]

1.5 SISTEME DE NUMERAȚIE NECONVENȚIONALE

Am discutat în subcapitolele anterioare depsre aritmetica convențională. În acest subcapitol se vor prezenta noțiunile de bază descre aritmetica neconvențională, precum și câteva avantaje ale folosirii acesteia într-un sistem ce are la bază operația de adunare.

În cazul aritmerticii convenționale am observat relația de dependență dintre transportul de intrare C_i de pe poziția i și C_{i+1}, în cazul sumatorului Ripple-Carry. În acest caz procedeul de adunare este pur secvențial, iar suma a două numere este realizată într-o perioadă de timp direct proporțională cu lungimea operanzilor. Pe de altă parte, în cazul sumatorului cu prefix paralel s-a făcut un compromis, renunțând la arie pentru rapiditatea de calcul. Cu toate acestea, relația de dependență între lungimea operanzilor și transport duce la imposibilitatea de a realiza o adunare complet paralelă folosind un sistem de numerație convențional.

Un sistem de numerație unde orice bit al semnalului de ieșire depinde doar de un număr redus de biți ai semnalului de intrare este cazul ideal în vederea îmbunătățirii performanței. Acest lucru este realizabil apelând la un sistem de numerație neconvențional unde propagarea transportului poate fi redusă sau chiar eliminată. Două astfel de posibilități sunt: sistem de numerație cu reziduuri sau reprezentarea ternară a numerelor binare.

1.5.1 SISTEM DE NUMERAȚIE CU REZIDUURI

Sistemul de numerație cu reziduuri (RNS) este un sistem capabil să prelucreze date foarte rapid, avantajele fiind posibilitatea de calcul paralel, fără transport și persmisiv la erori în aritmetică. Într-un sistem de numerație cu reziduuri există câteva numere prime care formează baza sistemului și ponderile pentru fiecare poziție. De exemplu, 5, 7 și 11 pot forma baza unui sistem RNS, iar pentru fiecare numar este reprezentat relativ la rezisuurile față de aceste numere din bază. Reprezentarea folosind acest sistem pentru numerele 217, 103, 320 este:

$$(217) \rightarrow \{217\ mod\ 5, 217\ mod\ 7, 217\ mod\ 11\} = \{2,0,8\} \quad (1.16)$$

$$(103) \rightarrow \{108\ mod\ 5, 108\ mod\ 7, 108\ mod\ 11\} = \{3,5,4\} \quad (1.17)$$

$$(320) \rightarrow \{320\ mod\ 5, 320\ mod\ 7, 320\ mod\ 11\} = \{0,5,1\} \quad (1.18)$$

Se observă că reprezentarea lui 320 este echivalentă cu suma reprezentărilor celor două numere: 217 și 103; (2 + 3) mod 5 = 0, (0 + 5) mod 7 = 5, (8 + 4) mod 11 = 1. [1]

Folosind RNS suma reziduurilor raportată la fiecare modul este independentă de reziduurile de la poziția următoare. Acest lucru reprezintă un avantaj al acestui sistem de numerație din punct de vedere al vitezei de operare a adunării. În cazul altor operații aritmetice precum împărțirea, radicalul detecție de semn modul de lucru este mai dificil în acest tip de sistem. De aceea, sistemul RNS este util în cadrul aplicațiilor unde aceste operații nu sunt necesare, sau se pot realiza într-o formă restransă, precum împărțirea printr-o constantă.

În general, apare nevoia de conversie din reprezentarea în RNS la o reprezentare folosind un sistem convențional, în vederea îndeplinirii eficiente a operațiilor folosind orice tip de operanzi. Principalele probleme asociate cu RNS sunt:

- Sistemul nu este complet și nu permite toate operațiile primitive
- Procesul de conversie din și în RNS este complicat și consumator de timp
- Implementarea hardware este mai complexa comparativ cu un sumator cu aritmetică convențională. [1]

1.5.2 ARITMETICĂ REDUNDANTĂ

Câteva dintre obiectivele folosirii aritmeticii redundante sunt:

- facilitarea implementării hardware
- reprezentarea numerelor într-o manieră compactă
- oferirea protecției împotriva erorilor

Reprezentarea numerelor folosind aritmetica redundantă se face folosind următoarea formula:

$$X = S_n \cdot 2^n + S_{n-1} \cdot 2^{n-1} + \cdots + S_0 \quad (1.19)$$

Termenii S_i , cu $i \in \{0,1 \dots, n\}$, iau valori din una din mulțimile $\{0,1,2,3\}$ sau $\{-1,0,1\}$. Acest lucru implică existența unor simboluri cărora le aparțin cel puțin 2 reprezentări.

Două dintre cele mai folosite tipuri de codări sunt codarea pozitiv-negativă și codarea pozitiv-negativ-complementată.

Codarea pozitiv negativă conține perechi de biți pozitivi și biți negativi. De exemplu, un număr, X, se poate reprezenta astfel:

$$X = 1 \leftrightarrow (x^1, x^0) = (1,0) \ (1.20)$$

$$X = 0 \leftrightarrow (x^1, x^0) = (0,0) \ (1.21)$$

$$X = -1 \leftrightarrow (x^1, x^0) = (0,1) \ (1.22)$$

Termenul x^1 corespunde biților pozitivi, iar termenul x^0 corespunde biților negativi. Relația dintre cei doi biți este următoarea:

$$X = x^1 - x^0 \ (1.23)$$

Se poate observa că perechea $(1,1)$ poate fi o reprezentare validă pentru valoarea numerică 0. Negarea biților în această codare este realizată prin interschimbarea valorilor x^1 și x^0. Rezultatul ne duce la un alt tip de codare, numită negativ-pozitivă.

Codarea pozitiv-negativ-complementată reprezintă un alt tip de codare cu doi biți folosită recent. De exemplu, un număr, X, se poate reprezenta astfel:

$$X = 1 \leftrightarrow (x^1, x^0) = (1,1) \ (1.24)$$

$$X = 0 \leftrightarrow (x^1, x^0) = (0,1)/(1,0) \ (1.25)$$

$$X = -1 \leftrightarrow (x^1, x^0) = (0,0) \ (1.26)$$

Termenul x^1 corespunde biților pozitivi, iar termenul x^0 corespunde biților negativi. Relația dintre cei doi biți este următoarea:

$$X = x^1 - \overline{x^0} \ (1.27)$$

Se poate observa că și acest tip de codare folosește două reprezentari pentru valoarea numerică 0, reprezentările fiind simetrice. Cu alte cuvinte, negarea reprezentării poate fi obținută interschimbând valorile termenilor x^1 și x^0.

1.6 EVOLUȚIA SUMATOARELOR FOLOSIND ARITMETICA REDUNDANTĂ

Sumatoarele ce folosesc aritmetica redundantă nu sunt unice, întrucât se folosesc diferite tipuri de codări. Abordarea implemntării conține o amprentă personală, dirijată de expresiile obținute și regulile ce trebuie respectate pentru operația de adunare corespunzătoare fiecărei

reprezentări. De-a lungul timpului, au ieșit în evidență numeroase implementări de sumatoare cu aritmetică redundantă. În continuare se vor prezenta două tabele ce ilustrează tipul de codare folosit și tehnologia de proiectare pentru fiecare sumator în parte.

Sumator implementat	Tipul de codare folosită
Harata et al [2]	Negativ-pozitiv
Kuniobu et al. [3]	Semn-valoare
Makino et al [4]	Pozitiv-negativ cu dublă reprezentare pentru zero
Kim et al. [5]	Pozitiv-negativ-complementat
Takahashi et al. [6]	Pozitiv-negativ
Sarrigeorides and Rabaey [7]	Pozitiv-negativ cu dublă reprezentare pentru zero
Savaș [8]	Pozitiv-negativ

Tabel 1.2 Tipuri de sumatoare și tipul de codare corespunzătoare [1]

Sumator implementat	Tehnologie
Harata et al [2]	$2.7 \mu um\ E/D - NMOS$
Kuniobu et al. [3]	$0.8\ \mu um\ CMOS$
Makino et al [4]	$0.5\ \mu um\ CMOS$
Kim et al. [5]	$0.35\ \mu um\ CMOS$
Takahashi et al. [6]	$1.2\ \mu um\ CMOS$
Sarrigeorides and Rabaey [7]	$0.25\ \mu um\ CMOS$
Savaș [8]	$0.35\ \mu um\ CMOS$

Tabel 1.3 Tipuri de sumatoare și tehnologia de proiectare [1]

2 PLACĂ DE DEZVOLTARE VC707 CU FPGA VIRTEX-7

2.1 FPGA – GENERALITĂȚI

Un FPGA (Field-programmable gate array) este o rețea de porți logice programabile, în general dispuse sub formă de matrice de blocuri programabile. Acestea sunt programate prin scrierea celulelor de memorie SRAM. Un FPGA are memorie volatilă – trebuie sa fie reprogramate la fiecare alimentare. Deși la început volatilitatea unui FPGA a fost considerată un dezavantaj pentru implementarea funcțiilor logice, în anii ’90, cercetătorii au realizat că volatilitatea circuitelor bazate pe SRAM poate să fie privită ca un avantaj pentru anumite aplicații, realizând astfel sisteme de calcul bazate pe FPGA cu performanțe comparabile cu cele alea supercomputerelor.

Un circuit FPGA este un circuit generic ce se poate programa în vederea implementării oricărei funcții definite de utilizator. Blocurile programabile pot îndeplini diverse funcții precum: generatoare de funcții logice, memorii de tip RAM sau ROM, circuite de înmulțire.

În general, FPGA-urile sunt folosite în sisteme de tip embedded și pentru calculatoare de înaltă performanță.

Avantajele utilizării unui FPGA

- Lucrează paralel – se pot implementa sisteme de calcul de înaltă performanță
- Sunt ideale pentru sisteme dedicate, producție de serie mică și protoipizare

Dezavantajele utilizării unui FPGA

- Costuri mari

Un FPGA este un dispozitiv electronic pe baza de semiconductori, sub formă de circuit integrat, care conține celule elementare de memorie și module elementare ce îndeplinesc funcții logice de bază (ȘI, SAU, SAU-exclusiv etc.).

Fiecare tip de circuit programabil conține: interconexiuni, comutatoare programabile și un bloc logic. Interconexiunile reprezintă resursele de conexiuni. Comutatoarele programabile permit conectarea elementelor de logică la firele de legături sau a firelor de legătură între ele. Blocul logic este un circuit de bază care este multiplicat în aria programabilă – este descompus în circuite mai mci care sunt mapate în blocul logic.

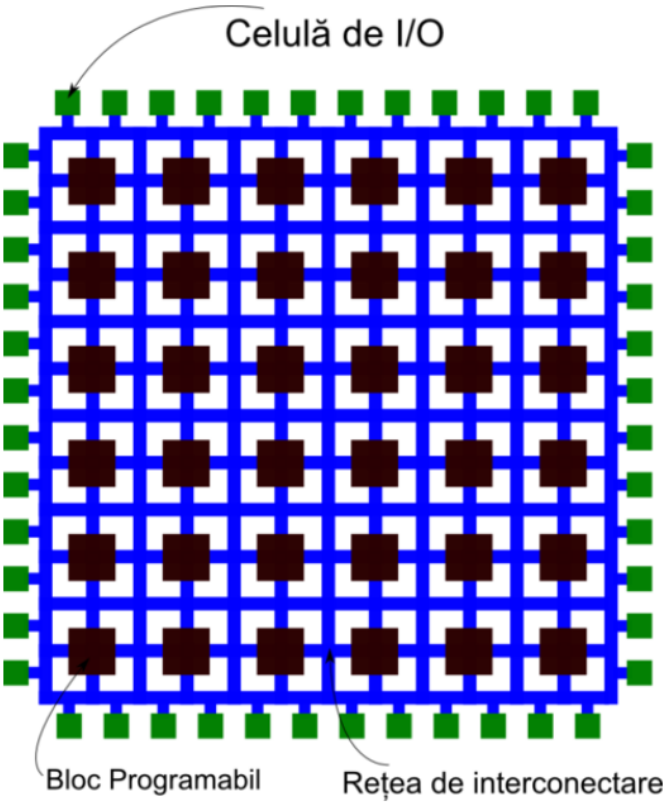

Fig. 2.1 Arhitectura unui FPGA [9]

Programarea unui FPGA necesită parcurgerea unor pași bine definiți în vederea realizării trecerii de la un limbaj de descriere hardware (Verilog, VHDL etc.) la un șir de biți necesar pentru configurarea cipului. Aceste etape sunt:

1. Sinteza – în cadrul acestei etape, codul în limbaj de descriere harware (VHDL) este interpretat de programul de sinteză care generează din acesta un circuit – un ansamblu de porți logice și registre – ce poartă denumirea de *netlist*; este singura etapă independentă de modelul de FPGA folosit.
2. Implementare - această etapă presupune translatarea circuitului la nivel de porți logice (*netlist*) în circuit la nivel de elemente disponibile pe cipul FPGA utilizat.
3. Generarea fișierului de programare *.bit* – folosind circuitul obținut în urma implementării, se generează un fișier cu extensia *.bit*; acest fișier conține șirul de biți necesar pentru configurarea blocurilor logice, blocurilor de intrare/ieșire și blocurilor de interconectare corespunzătoare circuitului dorit de utilizator.
4. Programarea – în ce-a de-a patra etapa, și ultima, printr-o interață JTAG, se va încarca fișierul *.bit* în memoria SRAM a cipului FPGA, realizându-se astfel configurarea dispozitivului [9]

Un FPGA se poate folosi sau este recomandat să fie folosit în aplicații precum: aplicații cu o rată de eșantionare ridicată (1-70 samples/s), aplicații cu o complexitate mare, sisteme de calcul în virgulă fixă sau mobilă, modemuri audio, video sau radio. Se pot observa în următorul grafic performanțele mai multor sisteme reconfigurabile precum microprocesoare, microcontrolere, procesoare digitale de semnal (DSP), circuite integrate specifice aplicației (ASIC). Graficul evidențiază rata de eșantionare obținută în funcție de complexitatea algoritmului folosit.

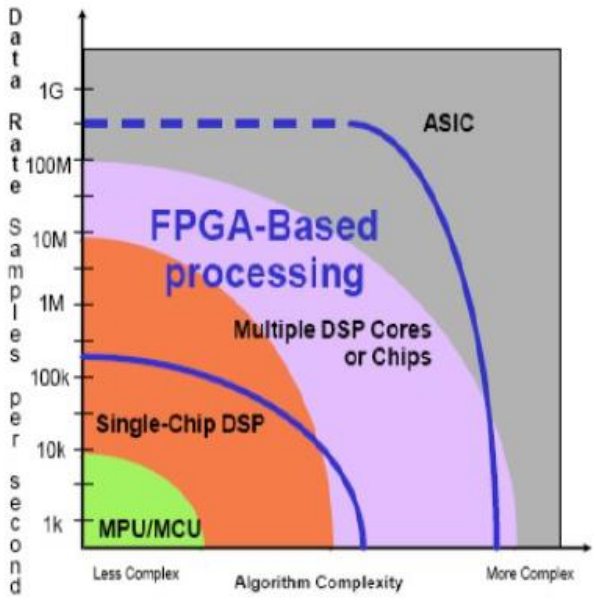

Fig. 2.2 Grafic performanță vs. complexitate [10]

Principalii producători de circuite de tip FPGA sunt: Xilinx, Altera, Actel, Lattice, Atmel.

La momentul actual, producătorii de circuite de tip FPGA urmăresc ca prin implementarea unui produs nou să îndeplinească următoarele cerințe:

- Sistemele să aibă un consum cât mai scăzut de putere
- Raportul calitate/preț să fie cât mai bun
- Creșterea eficienței utilizatorului în vederea dezvoltării unor aplicații, fără a se omite elemente inovatoare

2.2 XILINX – FAMILII DE CIRCUITE INTEGRATE DE TIP FPGA

Unul dintre principalii producători de circuite de tip FPGA este Xilinx. În momentul de față, Xilinx deține o gamă variată de familii de cipuri FPGA precum: Spartan, 7-Series (Virtex – 7, Kintex – 7, Artix- 7), 5-Series, 6 – Series (Kintex 6, Virtex 6, Artix 6).

În figura următoare este prezentată o comparație între câteva produse ale firmei Xilinx, din punct de vedere al specificațiilor precum: număr de celule logice, număr de pini de intrare/ieșire destinați folosirii de către utilizator, număr de standarde de intrare/ieșire suportate, tehnologie de management al ceasului, bloc de memorie RAM, existența unui bloc de înmulțire destinat procesoarelor digitale de semnal, viteza transmisiei seriale, existența magistralei PCI Express, posibilitatea de implementare a unui procesor software.

Features	Virtex-6	Virtex-5	Spartan-6	Extended Spartan-3A
Logic Cells	Up to 760,000	Up to 330,000	Up to 150,000	Up to 53,000
User I/Os	Up to 1200	Up to 1200	Up to 576	Up to 519 I/O
I/O Standards Supported	Over 40	Over 40	Over 40	Over 20
Clock Management Technology	PLL	DCM + PLL	DCM + PLL	DCM
Embedded Block RAM	Up to 38 Mbits	Up to 18 Mbits	Up to 4.8 Mbits	Up to 1.8 Mbits
Embedded Multipliers for DSP	Yes (25 x 18 MAC)	Yes (25 x 18 MAC)	Yes (18 x 18 MAC)	Yes (18 x 18 MAC)
Multi-Gigabit High Speed Serial	6.5 Gbps, beyond 11 Gbps	3.75 Gbps, 6.5 Gbps	3.125 Gbps	No
PCI Express® Technology	Gen 1, x8, hard Gen 2, x8, hard	Gen 1, x8, hard Gen 2, x8, soft	Gen 1, x1, hard	No
MicroBlaze™ Soft Processor	Yes	Yes	Yes	Yes

Fig. 2.3 Comparația unor specificații între diverse familii de FPGA [10]

În cadrul familiei 7 de FPGA (7 Series) produse de Xilinx avem 3 tipuri de circuite integrate FPGA: Artix-7, Kintex-7, Virtex-7. În figura următoare este evidențiat atât nivelul de performanță pentru fiecare model, cât și puterea consumată de acestea. Din figură se observă că Virtex-7 are cel mai ridicat nivel de performanță și cel mai scăzut consum de putere.

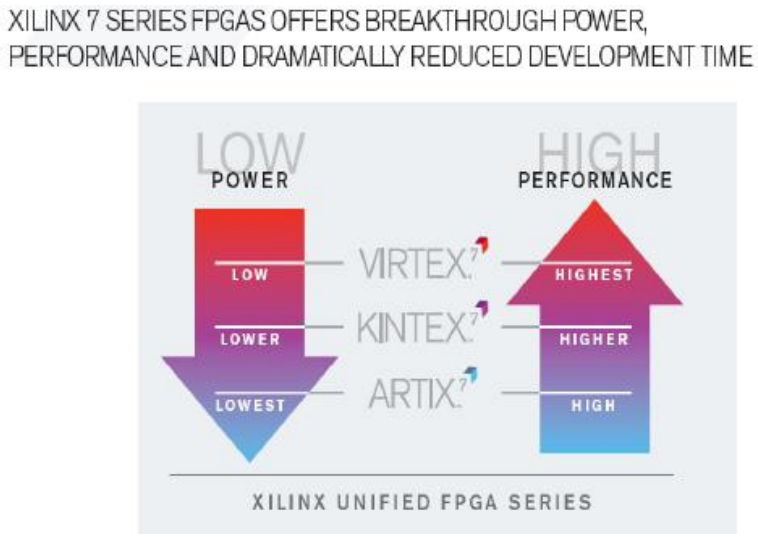

Fig. 2.4 Comparație între Virtex7, Kintex 7 și Artix 7 [10]

În urmatoarea figură este realizată o comparație între cele trei modele de cipuri FPGA din familia 7 (7 Series) produsă de firma Xilinx. Comparația este riguroasă și ține cont de următoarele criterii: număr de celule logice, memorie RAM, performanță DSP, număr de transmițătoare, viteza de transmisie, maximul vitezei transmisiei seriale, generația magistralei PCI Express, pini de intrare/ieșire, tensiune maximă suporatată de pinii de intrare/ieșire, tipul de aplicații vizate.

MAXIMUM CAPABILITY	ARTIX-7 FAMILY	KINTEX-7 FAMILY	VIRTEX-7 FAMILY
Logic Cells	352K	478K	1,955K
Block RAM	12Mb	34Mb	65Mb
DSP Slices	700	1,920	3,960
Peak DSP Performance (symmetric FIR)	504 GMACS	2,450 GMACS	5,053 GMACS
Transceiver Count	4	16	88
Peak Transceiver Speed	3.75Gbps	10.3125Gbps	28.05Gbps
Peak Serial Bandwidth (full duplex)	30Gbps	800Gbps	2,784Gbps
PCI Express® Interface	Gen1 x4	Gen2 x8	Gen3 x8
Memory Interface	1,066Mbps	2,133Mbps	2,133Mbps
I/O Pins	450	500	1,200
I/O Voltage	1.2V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V
Packaging Options	Low cost wire bond	Low cost lidless flip chip and High performance flip chip	Highest performance flip chip
Target Applications	• Handheld portable ultrasound • Digital SLR lens control module • Software defined radio	• Wireless LTE infrastructure • 10G PON OLT line card • LED backlit and 3D video displays • Medical imaging • Avionics imaging • Video-over-IP	• 400G and 100G line cards • 100G OTN muxponder • 100GE line card • 300G bridge • Terabit switch fabric • RADAR • ASIC emulation • High-performance computing • Test and measurement

Fig. 2.5 Comparație familii Virtex7, Kintex 7 și Artix 7 [10]

Ținând cond de performanța la care s-a ajuns în momentul de fața în producția de circuite integrate, fabricarea unui circuit integrat dedicat poate sa fie costisitoare. Acest lucru este direct influențat de tehnologia cu care se lucrează. Astfel, utilizatorii se orientează către sisteme reconfigurbile dezvoltate și testate, renunțând la implementarea pe cont propriu a unor circuite integrate. Acest lucru este evidențiat în graficul următor, unde se observă că odată cu reducerea consumului de putere se modifică și tehnologia cu care se lucrează în siliciu, lucru ce implică majorarea costurilor.

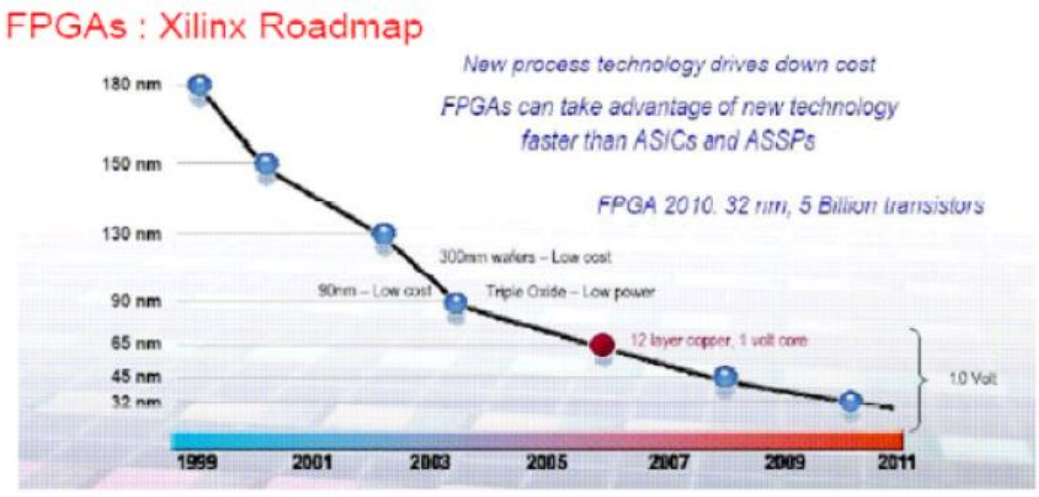

Fig. 2.6 Evoluția proceselor tehnologice de la aparația FPGA-urilor până în prezent [10]

2.3 ISE DESIGN SUITE

Xilinx, pe lângă producția de circuite integrate de tip FPGA, pune la dispoziție și medii de dezvoltare de tip HDL (Hardware Description Language), în vederea sintezei modulelor dezvoltate de utilizator, implementării și generării fișierului cu extensia .bit (bitstream) folosit pentru a programa FPGA-ul. Prin sinteză se înțelege compilarea modulului scris în limbaj de descriere hardware (HDL). Un astfel de mediu de dezvoltare software pus la dispoziție de Xilinx este Xilinx ISE (Integrated Synthesis Environment). Începând cu anul 2013, Xilinx a întrerupt dezvoltarea pentru Xilinx ISE, trecând la Vivado Design Suite. Acesta este un mediu de dezvoltare integrat (IDE). Pe langă funcționalitățile deținute și de ISE, Vivado Design Suite pune la dispoziție, în plus, un simulator, Vivado Simulator, care are avantajul de a suporta mai multe limbaje. [3]

Mediul de dezvoltare ISE Design Suite, pe lângă sinteza modulului implementat și generarea fișierului .bit util pentru programarea FPGA-ului, pune la dispoziție și multe alte lucruri. În primul rând, ISE Design Suite conține un program, iMPACT, folosit pentru programarea indirectă a memoriei Flash a FPGA-ului, pe baza fișierului .bit generat în urma proceselor de sinteză și implementare. Alte facilități oferite de mediul de dezvoltare sunt analiza de timp și performanță a modulului implementat, analiza comportamentală a semnalelor în interiorul modulelor, echivalent cu vizualizarea lor pe osciloscop, facilitate realizabilă prin intermediul programului Chipscope Pro.

De asemenea, ISE Design Tools integrează și un simulator, oferind posibilitatea analizei corectitudinii și performanțelor blocurilor înaintea implementării acestora.

2.4 FPGA VIRTEX-7 – SPECIFICAȚII TEHNICE

În cadrul acestui proiect se dorește implementarea unui înmulțitor rapid ce lucrează cu aritmetică redundantă, rezultatele urmând să fie evidențiate folosind o placă de dezvoltare cu cip FPGA Virtex-7. Motivul principal pentru care s-a ales aceasta famillie, 7 Series, și anume Virtex-7 este în vederea unei dezvoltari uletrioare a acestui modul de înmulțire, de la funcționarea cu operanzi în virgulă fixă, la funcționarea cu operanzi în virgulă mobilă.

În figura 2.7 este prezentată placa de dezvoltare VC707, cu toate elementele componente ce stau la dispoziția utilizatorului. În prezenta lucrare, de interes sunt afișajul LCD cu 2 linii și 16 coloane, puntea USB-to-UART, ce face posibilă comunicația serială între calculatorul utilizatorului și FPGA, cele 8 leduri, folosite pentru validarea transmisiei seriale, butoanele de tip push-button, folosite pentru activarea transmisiei seriale și pentru reset și interfața USB JTAG folosită pentru programarea și depanarea FPGA-ului.

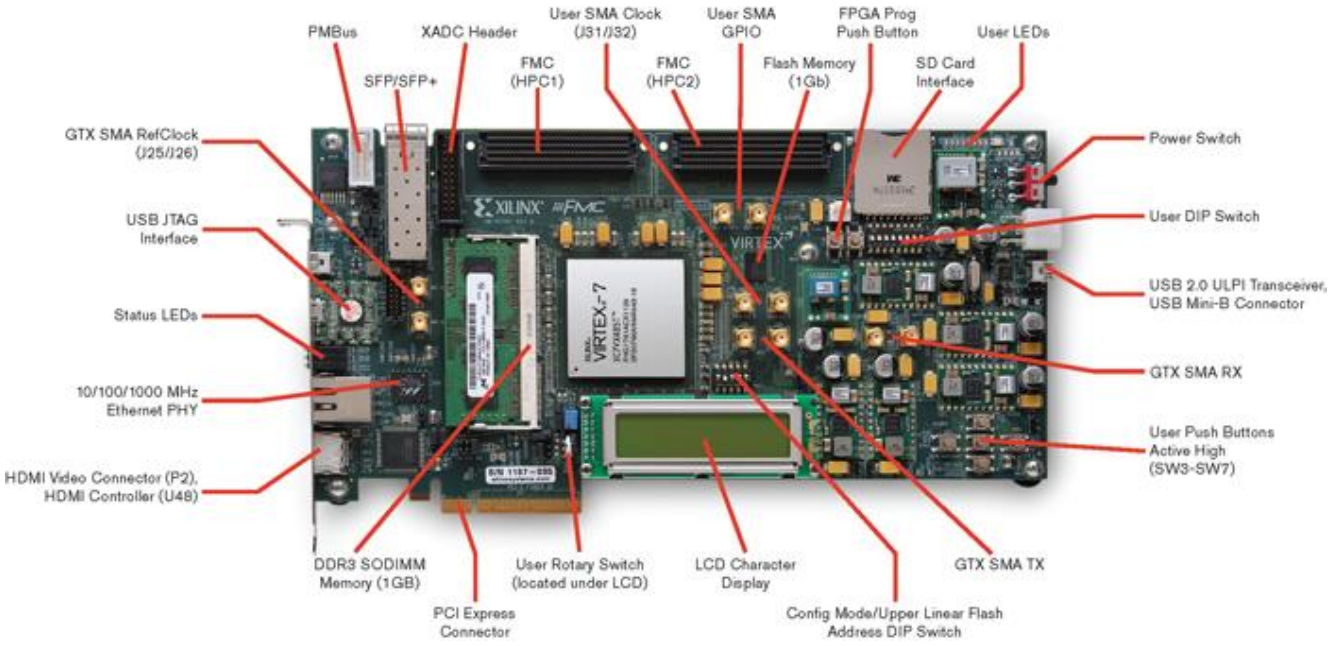

Fig. 2.7 Placă de dezvoltare VC707 cu FPGA Virtex 7 [11]

2.4.1 SEMNALUL DE CEAS AL SISTEMULUI

Placa de dezvoltare VC707 dispune de cinci surse de semnal de ceas. Semnalul de ceas al sistemului provine de la un oscilator de tip LVDS, cu frecvența fixă de 200 MHz. Oscilatorul prezină două semnale de ieșire de tip diferențial, SYSCLK_P și SYSCLK_N, conectate la cipul FPGA la pinii E19, respectiv E18. Figura 2.8 evidențiază schema electrică a acestui oscilator.

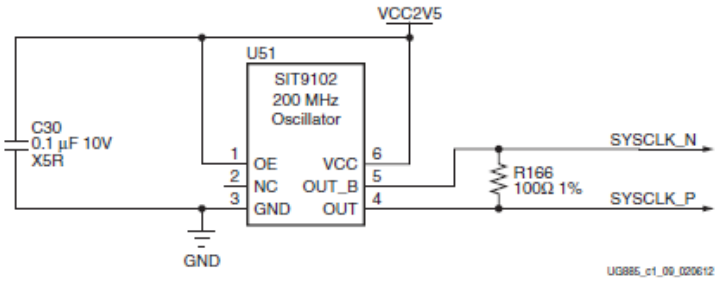

Fig. 2.8 Oscilator cu frecvența 200 MHz [11]

2.4.2 PORTURI INTRARE/IEȘIRE

Placa de dezvoltare VC707 pune la dispoziția utilizatorului următoarele porturi de intrare/ieșire, de uz general:

- 8 leduri: GPIO_LED[7:0]
- un comutator de reset și 5 butoane
- un dip-switch cu 8 poziții
- un switch rotativ
- un afișaj LCD cu 2 linii și 16 coloane

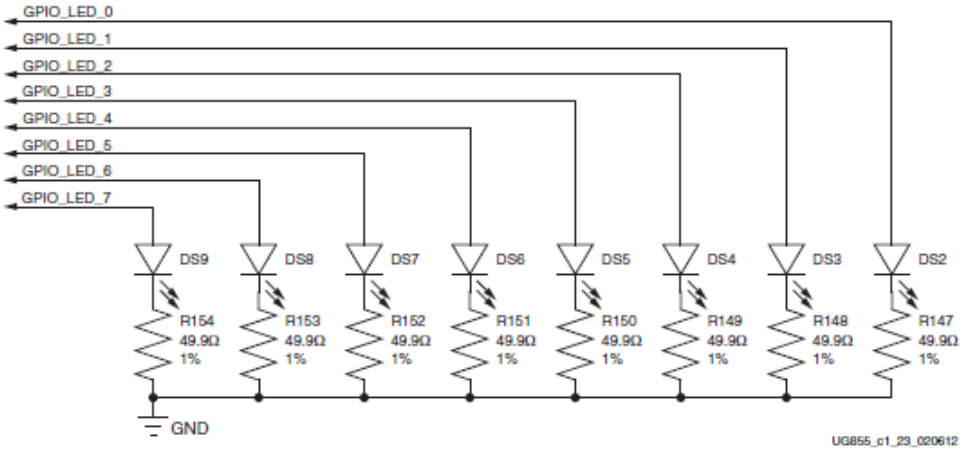

Fig. 2.9 Schema electrică de conectare a ledurilor [11]

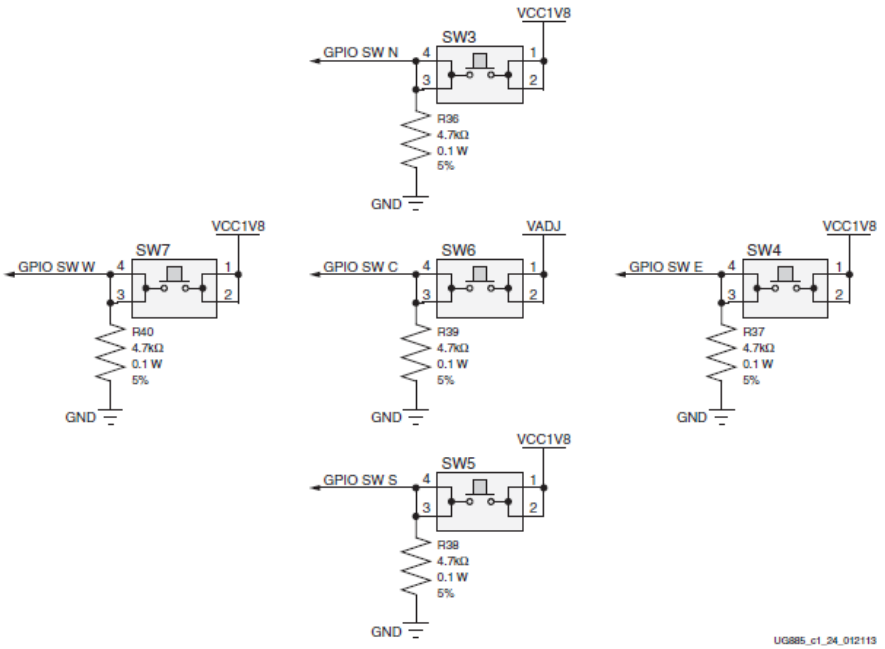

Fig. 2.10 Schema electrică de conectare a butoanelor [11]

2.4.3 CONTROLER LCD

2.4.3.1 Funcționare

Kit-ul de dezvoltare VC707 conține un afișaj LCD de 2 x 16 caracatere. Acest LCD dispune de un controler grafic intern, iar funcționalitățile acestuia vor fi prezentate în continuare.

Fig. 2.11 LCD display [12]

Folosirea unui afișaj LCD este o modalitate foarte utilă de a evidenția o varietate mare de informații folosind standardul ASCII și caractere definite de utilizator. Cu toate acestea, acest tip de afișaj nu este foarte rapid, perioada minimă de afișare fiind de 0.5s, aceasta fiind limita minimă pentru a asigura claritatea caracterelor afișate. Comparând cu ceasul de frecvență egală cu 200MHz, disponibil pe placă, afișajul este încet.

Afișajul LCD lucrează la o tensiune de 5V și este conectat la FPGA printr-un translator de nivel de tensiune bidirecțional de 8 biți.

În figura este evidențiat circuitul folosit pentru conectarea afișajului LCD.

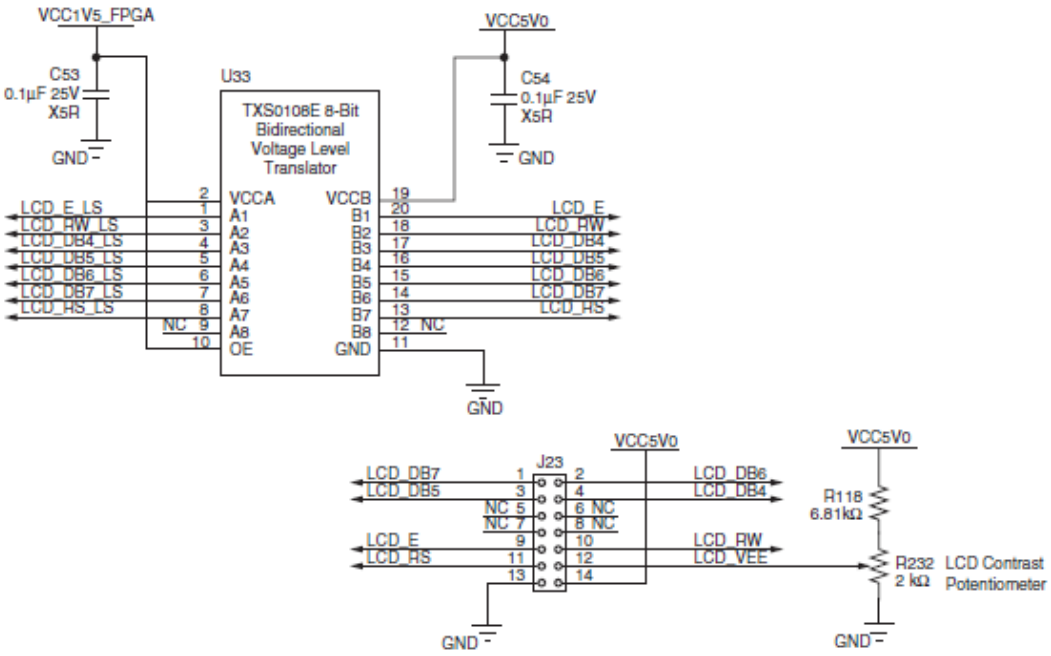

Fig. 2.12 Schema electrică de conectare a afișajului [10]

În următorul tabel sunt explicate semnalele interfeței afișajului.

Nume semnal	Pin FPGA	Funcție	
LCD_DB<3>	AN40	Bit de date DB3	
LCD_DB<2>	AR39	Bit de date DB2	
LCD_DB<1>	AR38	Bit de date DB1	
LCD_DB<0>	AT42	Bit de date DB0	
LCD_E	AT40	Semnal de activare citire/scriere 0: Dezactivat 1: Operația de citire/scriere este activată	
LCD_RS	AN41	Semnal de selectare registru 0: Registru de instrucțiuni în timpul operațiilor de citire 1: Registrul de date pentru operațiile de citire sau scriere	
LCD_RW	AR42	Semnal de control citire/scriere 0: Scriere, LCD accepta date 1: Citire, LCD trimite date	

Tabel 2.1 Semnalele interfeței afișajului lcd [12]

2.4.3.2 Maparea memoriei

2.4.3.2.1 DD RAM

Controlerul grafic conține trei regiuni de memorie, fiecare având un scop specific: DD RAM, CG ROM si CG RAM. Înainte de a accesa oricare din aceste regiuni de memorie, este necesară inițializarea afișajului.

Memoria DD RAM stocheaza codul caracterului ce urmează să fie afișat pe ecran. Majoritatea aplicațiilor interacționează cu memoria DD RAM. Caracterul codului stocat în DD RAM face referință fie la un caracter din setul de caractere al memoriei predefinite CG ROM, fie setat de utilizator și stocat în memoria CG RAM.

În figură se poate observa adresa predefinita pentru fiecare din cele 32 de locații disponibile pe afișaj. Prima linie de caractere este stocată între adresele 0x00 și 0x0F, iar a doua linie de caractere este stocată între adresele 0x40 și 0x4F.

	Character Display Addresses																Undisplayed Addresses		
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	…	27
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	…	67
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	40

Fig. 2.13 Adresele locațiilor din memoria DD RAM [12]

La nivel fizic însă, sunt 80 de locații de memorie în DD RAM, folosite pentru stocarea caracterelor, câte 40 de caractere pe fiecare linie. Locațiile de la adresa 0x10 până la adresa 0x27 și de la adresa 0x50 până la adresa 0x67 pot fi folosite pentru stocarea unor date neafișate. Alternativ, aceste locații pot stoca și caractere ce vor fi afișate folosind funcțiile de shiftare ale controlerului grafic.

Comanda „Set DD RAM Address” inițializează adresa numărătorului înainte de scrierea sau citirea memoriei. Scrierea memoriei DD RAM se realizează folosind comanda „Write Data to CG RAM or DD RAM”, iar scrierea se realizează folosind comanda „Read Data from CG RAM or DD RAM”.

Adresa numărătorului memoriei DD RAM are posibilitatea de a rămîne constantă după fiecare citire sau scriere sau să se auto-incrementeze cu câte o locație. Acest lucru este determinat de setarea sau resetarea bitului I/D corespunzator comenzii „Entry Mode Set”.

2.4.3.2.2 CG ROM

Memoria CG ROM conține caracterele predefinite pe care afișajul le poate folosi. Astfel, codul caracterului stocat în memoria DD RAM pentru fiecare locație a caracterului face referință către o poziție din memoria CG ROM. De exemplu, un cod de caracter pentru o cifra în hexazecimal de 0x53, stocat într-o locație din memoria DD RAM, va afișa caracterul ‚S’. Niblul superior de date, corespunzator lui 0x53, are, în binar, valoarea DB[7:4]=0101, iar niblul inferior de date, are, în binar, valoarea DB[3:0]=0011. Caracterele corespunzatoare alfabetului roman sunt stocate în memoria CG ROM la adresele corespunzatoare codului în ASCII al fiecărui caracter. [12]

În figura 2.14 este ilustrat setul de caractere predefinit stocat în memoria CG ROM. Adresa corespunzatoare din memorie, la care sunt stocate caracterele este formată din niblul superior, ilustrat pe coloane concatenat cu niblul inferior, ilustrat pe linii.

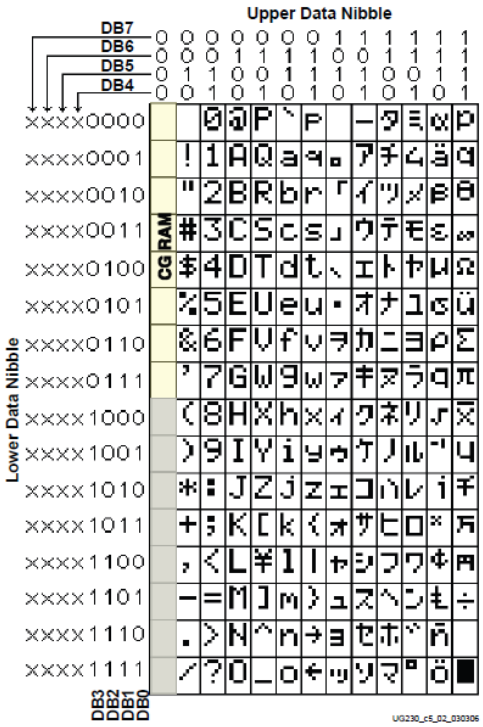

Fig. 2.14 Codul ASCII pentru caractere stocat în memoria CG ROM [12]

2.4.3.2.3 CG RAM

Memoria CG RAM dispune de spațiu în vederea creării a opt caractere la alegerea utilizatorului. Fiecare locație permite crearea unui caracter format într-o matrice de puncte cu 8 linii și 5 coloane, așa cum este evidențiat în figura 2.14.

						Upper Nibble				Lower Nibble			
						Write Data to CG RAM or DD RAM							
A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
Character Address			Row Address			Don't Care			Character Bitmap				
0	1	1	0	0	0	-	-	-	0	■	0	■	0
0	1	1	0	0	1	-	-	-	■	0	■	0	■
0	1	1	0	1	0	-	-	-	0	■	0	■	0
0	1	1	0	1	1	-	-	-	■	0	■	0	■
0	1	1	1	0	0	-	-	-	0	■	0	■	0
0	1	1	1	0	1	-	-	-	■	0	■	0	■
0	1	1	1	1	0	-	-	-	0	■	0	■	0
0	1	1	1	1	1	-	-	-	0	0	0	0	0

Fig. 2.15 Exemplu de caracter realizat de utilizator [12]

2.4.3.3 *Setul de comenzi*

În tabel sunt enumerate comenzile controlerului grafic și definirea biților corespunzători. Deoarece afișajul este setat pentru primirea operațiilor pe 4 biți de date, corespunzator modului de lucru pe 4 biți, fiecare comandă de 8 biți este trimisă ca 2 operații pe niblu. Niblul superior se va transfera primul, urmat de niblul inferior.

Function	LCD_RS	LCD_RW	Upper Nibble				Lower Nibble			
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Clear Display	0	0	0	0	0	0	0	0	0	1
Return Cursor Home	0	0	0	0	0	0	0	0	1	-
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S
Display On/Off	0	0	0	0	0	0	1	D	C	B
Cursor and Display Shift	0	0	0	0	0	1	S/C	R/L	-	-
Function Set	0	0	0	0	1	0	1	0	-	-
Set CG RAM Address	0	0	0	1	A5	A4	A3	A2	A1	A0
Set DD RAM Address	0	0	1	A6	A5	A4	A3	A2	A1	A0
Read Busy Flag and Address	0	1	BF	A6	A5	A4	A3	A2	A1	A0
Write Data to CG RAM or DD RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0
Read Data from CG RAM or DD RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0

Fig. 2.16 Setul de comenzi pentru afișaj [12]

2.4.3.3.1 Ștergere afișaj (Clear Display)

Rolul funcției este de a șterge ce a fost afișat pe ecran și de întoarcere a cursorului în poziția inițială, pe prima linie și prima coloana a afișajului. Comanda scrie un spațiu (adresa ASCII pentru spațiu este 0x20) la toate adresele din DD RAM. Numărătorul de adrese al memoriei este resetat la 0, la locația 0x00. De asemenea, comanda șterge toate setările făcute anterior. Bitul I/D al comenzii „Entry Mode Set” este setat cu 1. Timpul de execuție necesar pentru această comandă este de $82\mu s - 1.64\ ms$.

2.4.3.3.2 Resetare cursor (Return Cursor Home)

Comanda de resetare cursor întoarce cursorul în poziția inițială, pe linia 1 și coloana 1. De asemenea, comanda întoarce și afișajul în poziția inițială, în eventualitatea în care a avut loc o deplasare a acestuia. Timpul de execuție necesar pentru această comandă este de $40\mu s - 1.6\ ms$.

2.4.3.3.3 Setare mod introducere (Entry Mode Set)

Funcția „Entry Mode Set” setează sensul de mișcare al cursorului și specifică dacă afișajul se va deplasa sau nu. Aceste operații se vor executa în timpul scrierii și citirii datelor. Timpul de execuție necesar pentru această comandă este de $40\mu s$.

Bitul de date DB1: (I/D) Incerementare/Decrementare

0	Adresa numărătorului de adrese se auto-decrementează. Cursorul se deplasează spre stânga.
1	Adresa numărătorului de adrese se auto-incrementează. Cursorul se deplasează spre dreapta.

Bitul de date DB0: (S - Shift) Deplasare

0	Deplasarea este dezactivată.
1	În timpul scrierii din memoria DD RAM, se deplasează întregul afișaj în direcția indicată ce bitul I/D.

2.4.3.4 Stingere/Aprindere afișaj (Display On/Off)

Rolul comenzii este de a stinge sau aprinde afișajul. Cursorul și poziția indicată de cursor clipesc. Timpul de execuție necesar pentru această comandă este de $40\mu s$.

Bitul de date DB2: (D – Display On/Off) Afișaj stins/aprins

0	Nu este afișat niciun caracter, însă datele stocate în memoria DD RAM se păstrează.
1	Sunt afișate datele stocate în memoria DD RAM.

Bitul de date DB1 (C) Cursor

0	Cursorul nu este afișat.
1	Cursorul este afișat.

Bitul de dte DB0: (B – Cursor blink On/Off) Clipire cursor

0	Cursorul nu clipește.
1	Cursorul clipește cu o perioadă de aproximativ jumătate de secundă.

2.4.3.4.1 Setare funcții (Function Set)

Comanda „Function Set” poate seta interfața cu care lucrează afișajul (dacă se folosesc 4 biți de date sau 8 biți de date), câte linii se folosesc și dimensiunea caracterelor. Cu toate acestea, controlerul disponibil suportă o singură setare, cu o valoare de 0x28. Timpul de execuție necesar pentru această comandă este de $40\mu s$. [12]

2.4.4 TRANSMISIA SERIALĂ A DATELOR

Componenta principală a unui port serial este un circuit UART (Universal Asynchronous Receiver/Transmiter). Pentru interacționarea unui calculator într-o comunicație serială, circuitul UART este obligatoriu, deoarece acest circuit realizează conversia datelor paralele de la calculator în formatul necesar pentru transmisia serială și conversia datelor seriale recepționate în formatul paralel utilizat de calculator.

Totodată circuitul adaugă bitul de start, bitul de stop și bitul de paritate la datele seriale transmise și detectează acești biți în cadrul datelor seriale recepționate.

Porturi seriale:

Porturile seriale au fost utilizare inițial pentru dispozitive care comunică bidirecțional cu un sistem. Acestea mai sunt cunoscute și sub numele de porturi de comunicație sau COM. Printre dispozitivele care pot comunica bidirecțional cu un sistem prin intermediul porturilor seriale amintim modem-ul, mouse-ul, scanner-ul care trimit și recepționează informații de la un calculator.

Portul serial transmite un „1” logic în intervalul -3 până la -25V și „0” logic în intervalul +3 până la +25V. Prin urmare portul serial are diferența maximă de semnal egală cu 50V. Din această

valoare reiese faptul că problema pierderilor de semnal nu este așa de gravă, rezultând astfel lungimi ale cablurilor de comunicație serială destul de mari.

Componentele sistemului de comunicație sunt următoarele:

- DTE – Data Terminal Equipment (calculatoare, terminale de data). Conțin interfețe seriale și controlere de comunicație.
- DCE – Data Communications Equipment. Aceste echipamente se numesc modemuri și permit calculatorului să transmită informații printr-o linie telefonică analogică. Funcțiile principale realizate de un modem sunt următoarele:
  - Conversia digital-analogică a informațiilor din calculator și conversia analog-digitală a semnalelor de la centrala telefonică analogică.
  - Modularea si demodularea unui semnal purtător. La transmisie, modemul suprapune (modulează) semnalele digitale ale calculatorului peste semnalul purtător al liniei telefonice. La recepție, modemul extrage (demodulează) informațiile transportate de semnalul purtător și le transferă calculatorului. [13]

În funcție de numărul de echipamente interconectate, o legătură serială poate fi punct la punct (legătură formată din două echipamente), sau multi-punct (legătură formată din mai mult de două echipamente).

Parametrii comunicaţiei seriale

Viteza de comunicație (numită și debit binar) este măsurată în biți/s (bps): D=1/T [biți/s], unde T este perioada de timp necesară pentru transmisia sau recepția unui singur bit.

Modemul reprezintă semnalele de date prin diferite stări electrice, în funcție de tipul de modulație pe care îl utilizează: frecvență, amplitutide sau fază. Fiecare stare electrică este menținută la ieșirea modemului pentru un interval de timp numit perioadă de modulație(Δ). [14]

Viteza de modulație este Vm = 1/ Δ [baud]. Unitatea de măsură a vitezei de modulație este baud după numele ingineerului și telegrafistului francez Jean-Maurice Baudot.

Relația dintre viteza de comunicație D și viteza de modulație Vm este: D= Vm*log2n [biți/s], unde n este numărul stărilor electrice distincte ale modemului. În cazul particular când există doar două stări electrice ale modemului viteza de comunicație este egală cu viteza de modulație, dar în general există un număr mai mare de stări electrice ale modemului, astfel încât viteza de comunicație devine multiplu al vitezei de modulație. [14]

Viteza de modulație este confundată adesea cu viteza de comunicație (debitul binar). Viteza de modulație este rata cu care se modifică stările electrice ale modemului într-o secundă.

Tipuri de comunicație serială

Din punct de vedere al direcției de transfer se pot remarca trei tipuri de comunicație serială:

- Simplex
- Semiduplex
- Duplex

În cazul comunicației simplex datele sunt transferate într-o singură direcție, de la un echipament transmițător la un alt echipament receptor. La comunicația semidulex, ambele echipamente care formează comunicația sunt pe rând echipament transmițător cât și echipament receptor. Pentru acest tip de comunicație este necesară o singură linie de transmisie formată din două fire de legătură. Într-o comunicație duplex (denumită și duplex integral sau full-duplex), datele se transferă simultan în ambele direcții. Primele conexiuni duplex necesitau două linii de transmisie (patru fire de legătură), ulterior s-a ajuns să se folosească o singură linie.

Din punct de vedere al sincronizării dintre transmițător și receptor, există două tipuri de comunicație serială:

- Asincronă
- Sincronă

Controlul fluxului de date

Controlul fluxului de date se realizează prin implementarea unor semnale speciale. Acest lucru a fost proictat pentru a fi posibilă comunicația între dispozitive cu viteze diferite. În momentul transmiterii datelor echipamentul receptor primește datele într-un buffer. În momentul umplerii acestuia echipamentul receptor trimite un semnal către transmițător pentru a întrerupe transmisia pentru golirea bufferului, moment în care echipamentul receptor transmite un semnal pentru a se relua transmisia. Această metodă este denumită metodă hardware.

Există și o metodă software, care se bazează pe transmiterea unor caractere speciale între două echipamente. În momentul în care echipamentul receptor, de exemplu o imprimantă, ajunge în punctul în care are bufferul plin, acesta trimite un caracter special echipamentului transmițător, de exemplu, un calculator. Atunci când receptorul poate primii din nou date transmite un alt caracter special pentru a notifica transmițătorul că poate relua transmiterea datelor.

Metoda de control se poate selecta prin intermediul driverului controlerului serial, sau prin intermediul unui comutator. Este foarte important ca amblele echipamente să folosească aceeași metodă de control pentru a se împiedica pierderile de date.

Metoda de control hardware

Metoda de control hardware presupune utilizarea unui protocol de comunicație cu ajutorul semnalelor de control ale interfeței seriale. Acest protocol se bazează pe stabilirea conexiunii între două echipamente și menținerea fluxului de date dintre acestea cât timp conexiunea este activă. Se folosesc în principal semnalele RTS și CTS. [14]

Metoda de control software

Metoda de control software presupune transmiterea unor caractere de control între cele două echipamente. În momentul în care echipamentul periferic nu mai poate primii date, trimite un caracter special pentru ca echipamentul transmițător să oprească transmiterea, iar în momentul în care perifericul poate primii din nou date transmite un alt caracter special pentru a informa transmițătorul că poate relua transmisia. Există două variante ale utilizării acestei metode. Prima se referă la folosirea caracterelor XON/XOFF, iar în a doua se folosesc caracterele ETX (End of TeXt)/ACK (ACKnowledge).

Folosind prima metodă echipamentul periferic va transmite XOFF în momentul în care acesta nu mai poate primii date, iar atunci când se poate relua transmisia va transmite caracterul XOFF.

Dacă se utilizează metoda a doua echipamentului periferic va transmite ETX pentru oprirea transmisie, urmat de ACK în momentul reluării transmisiei. [14]

3 ARITMETICĂ REDUNDANTĂ

3.1 REPREZENTARE FOLOSIND ARITMETICA REDUNDANTĂ

Reprezentarea numerelor folosind aritmetica redundantă se face folosind următoarea formula:

$$X = S_n \cdot 2^n + S_{n-1} \cdot 2^{n-1} + \cdots + S_0 \quad (3.1)$$

Termenii S_i , cu $i \in \{0,1 \ldots, n\}$, iau valori din una din mulțimile $\{0,1,2,3\}$ sau $\{-1,0,1\}$. Acest lucru implică existența unor simboluri cărora le aparțin cel puțin 2 reprezentări.

Unul din avantajele majore al folosirii aritmeticii redundante este crearea unor „buzunare" ce oferă posibilitatea stocării transportului (carry save). Prin folosirea aritmeticii redundante nu vom mai avea propagarea transportului, astfel, adunarea devine independetă de lungimea operanzilor, lucru care nu era posibil în cazul folosirii aritmeticii convenționale.

În aritmetica redundanta există numeroase tipuri de codari. În cazul in care se folosește mulțimea de valori $\{-1,0,1\}$, vom avea doi biți pentru codare: X_1 $și$ X_0.

Posibiliățile de codare sunt prezentate în tabelul 3.1:

Nr.		Value X (x1, x0)				Value RB (ax1 + bx0 + c)	
		00	01	10	11		
0	EN0	0	1	-1	NU(0)	x0 - x1	OK
1	EN0 DB	0	-1	1	NU(0)	x1 - x0	OK
2	EN1	-1	1	0	NU(2)	x1 + 2x0 -1	not OK
3	EN1 DB	-1	0	1	NU(2)	2x1 + x0 -1	not OK
4	EN2	1	0	-1	NU(-2)	1 - 2x1 - x0	not OK
5	EN2 DB	1	-1	0	NU(-2)	1 - x1 -2x0	not OK
6	EN3	0	1	NU(-2)	-1	x0 - 2x1	not OK
7	EN4	0	-1	NU(2)	1	2x1 - x0	not OK
8	EN5	-1	1	NU(-2)	0	2x0 - x1 - 1	not OK
9	EN6	-1	0	NU(0)	1	x1 + x0 +1	OK
10	EN7	1	0	NU(0)	-1	1 - x1 - x0	OK
11	EN8	1	-1	NU(2)	0	x1 - 2x0 +1	not OK
12	EN4 DB	0	NU(2)	-1	0	2x0 - x1	not OK
13	EN3 DB	0	NU(-2)	1	-1	x1 - 2x0	not OK
14	EN6 DB	-1	NU(0)	0	1	x1 + x0 -1	OK
15	EN5 DB	-1	NU(-2)	1	0	2x1 - x0 -1	not OK
16	EN8 DB	1	NU(2)	-1	0	x0 - 2x1 + 1	not OK
17	EN7 DB	1	NU(0)	0	-1	1 - x1 - x0	OK
18	EN9	NU(-2)	0	-1	1	x1 + 2x0 -2	not OK
19	EN10	NU(2)	0	1	-1	2 - x1 - 2x0	not OK
20	EN9 DB	NU(-2)	-1	0	1	2x1 + x0 - 2	not OK
21	EN11	NU(0)	-1	1	0	x1 - x0	OK
22	EN11 DB	NU(0)	1	-1	0	x0 - x1	OK
23	EN10 DB	NU(2)	1	0	-1	2 - 2x1 - x0	not OK
		PN	EN0, EN0 DB, EN11, EN11 DB				
		PNC	EN6, EN7, EN6 DB, EN7 DB				

Tabel 3.1 Posibilități de codare folosind aritmetica redundantă

Utilizând mulțimea de valori posibile $\{-1,0,1\}$ obținem o multitudine de posibilități de codare cu doi biți. În prezenta lucrare vom discuta despre două codari: codare pozitiv – negativ (PN) și codare pozitiv – negativ – complementat (PNC).

3.2 CODARE POZITIV – NEGATIV

În cazul codării pozitiv – negativ vom avea doi biți de pentru codare Z_1 *și* Z_0 și mulțimea de valori $\{-1,0,1\}$. Astfel, vor exista două codari pentru 0 și câte o singura codare pentru 1 și -1.

Codarea pozitiv negativă conține perechi de biți pozitivi și biți negativi. De exemplu, un număr, X, se poate reprezenta astfel:

$$X = 1 \leftrightarrow (x^1, x^0) = (1,0) \ (3.2)$$

$$X = 0 \leftrightarrow (x^1, x^0) = (0,0) \ (3.3)$$

$$X = -1 \leftrightarrow (x^1, x^0) = (0,1) \ (3.4)$$

Termenul x^1 corespunde biților pozitivi, iar termenul x^0 corespunde biților negativi. Relația dintre cei doi biți este următoarea:

$$X = x^1 - x^0 \ (3.5)$$

Pentru a obține valoarea în codare pozitiv – negativ, corespunzătoare unui numar binar, este necesară realizarea operației de scădere. Acest lucru este evidențiat în figura 3.1, unde sunt exemplificate mai multe transformări din reprezentare binară în codare pozitiv – negativă.

	a	**0**	**0 1**	**1**				**Codare:**				
	b	1	0 1	0				00 -> -0				
	RB	-1	0	1				11 -> 0				
	a-b	-1	0	1				10 -> 1				
								01 -> -1				
	a-b = RB										**Z = Z1 - Z0**	
	(RB)SB e sign bit in RB corectat (+1)										**Z = (Z1, Z0)**	
		Sa	**pos**	**neg**							**-Z = (Z0, Z1)**	
			0	**1**	**Sb**							
		pos	0 in RB	1 in RB		**Sign bit**						
		0	x (1)	1		RB = a - b						
			x (1)	0		(RB)a = not Sa						
		neg	-1 in RB	0 in RB		(RB)b = not Sb						
		1	0	x (0)								
			1	x (0)								
Caut sa generez produsele partiale sub forma a-b pentru a le transfera direct in RB. Complementarea algebrica se face in cazul PN inversand compomentele.												

Fig. 3.1 Codare pozitiv negativ – reprezentare, bit de semn

Bitul de semn în aritmetică redundantă este sintetizat folosind o diagrama Karnaugh evidențiată în figura 3.1.

În figura 3.2 este exemplificată operația decodare a unui număr, 147. Acesta este reprezentat folosind formula (3.1).

	1	1	1	0	0	1	0	1	0	202	minus
	1	0	0	1	1	0	1	1	1	55	
147	0	128	64	-32	-16	8	-4	0	-1	147	

Tabel 3.2 Exemplificarea unei operații de scădere folosind 2 operanzi codați PN

În figura 3.3 este ilustrată diagrama Karnaugh din care s-au sintetizat ecuațiile corespunzătoare semnalelor de intrare pentru cei doi operanzi, X și Y și pentru transporturile asociate acestora. Atât operanzii, cât și transportul sunt reprezentate folosind codarea PN. Se poate observa din figură că pentru fiecare căsuță din diagrama avem una sau mai multe posibilități de reprezentare. Fiecare căsuță din diagramă are următorul conținut: pe prima linie este semnalul de ieșire, Z, codat PN, iar pe cea de-a doua linie este transportul de ieșire, C_y, de asemenea codat PN. Transportul de ieșire nu poate sa fie total independent de cel de intrare. Astfel, se va încerca ca doar C_{y1}să fie independent de transportul de intrare. În acest fel, pe coloanele 0001, 0111, 0101, 0100 și 1101 vom avea $C_{y0} =' 1' și \ C_{y1} =' 0'$, iar pe coloanele 0010, 1110, 1011 și 1000 vom avea $C_{y0} =' 1' și \ C_{y1} =' 0'$.

	Cin1Cin0	0	-1	0	1	0	-1	-2	-1	0	-1	0	1	2	1	0	1
X1X0Y1Y0		0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
	0	0	-1	0	1	0	-1	-2	-1	0	-1	0	1	2	1	0	1
	00	00	01 10	00	10 01	00	01 10	00	01 10	00	01 10	00	10 01	00	10 01	00	10 01
		aa	aa 01	aa	aa 10	aa	aa 01	01	aa 01	aa	aa 01	aa	aa 10	10	aa 10	aa	aa 10
	-1	-1	-2	-1	0	-1	-2	-3	-2	-1	-2	-1	0	1	0	-1	0
	01	01 10	00	01 10	00	01 10	00	01	00	01 10	00	01 10	00	10 01	00	01 10	00
		aa 01	01	aa 01	aa	aa 01	01	01	01	aa 01	01	aa 01	aa	aa 10	aa	aa 01	aa
	0	0	-1	0	1	0	-1	-2	-1	0	-1	0	1	2	1	0	1
	11	00	01 10	00	10 01	00	01 10	00	01 10	00	01 10	00	10 01	00	10 01	00	10 01
		aa	aa 01	aa	aa 10	aa	aa 01	01	aa 01	aa	aa 01	aa	aa 10	10	aa 10	aa	aa 10
	1	1	0	1	2	1	0	-1	0	1	0	1	2	3	2	1	2
	10	10 01	00	10 01	00	10 01	00	01 10	00	10 01	00	10 01	00	10	00	10 01	00
		aa 10	aa	aa 10	10	aa 10	aa	aa 01	aa	aa 10	aa	aa 10	10	10	10	aa 10	10
	Nu pot fi ambele Cy1/0 independente de Cin1/0 pentru ca pe aceeasi coloana exista si "aa" si "01" sau "10". Incerc sa-l fac independent pe Cy1.														Cy0 = '1'; Cy1 = '0'		
	Pe coloanele 0001, 0111, 0101, 0100 si 1101 " a" trebuie sa fie ='0' iar pe 0010, 1110, 1010, 1011 si 1000 "a" trebuie sa fie '1' :														Cy0 = '0'; Cy1 = '1'		

Fig. 3.2 Diagramă Karnaugh - obținerea ecuațiilor pentru codarea PN

3.3 CODARE POZITIV – NEGATIV – COMPLEMENTAT

În cazul codării pozitiv – negativ vom avea doi biți de pentru codare $Z_1 \ și \ Z_0$ și mulțimea de valori $\{-1,0,1\}$. Astfel, vor exista două codari pentru 0 și câte o singura codare pentru 1 și -1.

Codarea pozitiv-negativ-complementată reprezintă un alt tip de codare cu doi biți folosită recent. De exemplu, un număr, X, se poate reprezenta astfel:

$$X = 1 \ \leftrightarrow (x^1, x^0) = (1,1) \ (3.6)$$

$$X = 0 \ \leftrightarrow (x^1, x^0) = (0,1)/(1,0) \ \ (3.7)$$

$$X = -1 \ \leftrightarrow (x^1, x^0) = (0,0) \ (3.8)$$

Termenul x^1 corespunde biților pozitivi, iar termenul x^0 corespunde biților negativi. Relația dintre cei doi biți este următoarea:

$$X = x^1 - \overline{x^0} \quad (3.9)$$

Pentru a obține valoarea în codare pozitiv – negativ – complementat, corespunzătoare unui numar binar, este necesară realizarea operației de scădere, urmată de adunarea rezultatului obținut cu 1. Acest lucru este evidențiat în figura 3.4, unde sunt exemplificate mai multe transformări din reprezentare binară în codare pozitiv – negativ – complementată.

a	**0**	**0 1**	**1**		**Codare:**		
b	0	1 0	1		00 -> -1		
RB	-1	0	1		01 -> 0		
a+b	0	1	2		10 -> 0		
					11 -> 1		
a+b = RB + 11...1 = (RB)SB - 1					**Z = Z1 - (notZ0) = Z1 + Z0 + 1**		
(RB)SB e sign bit in RB corectat (+1)					**Z = (Z1, Z0)**		
	Sa	**pos**	**neg**		**-Z = (notZ1, notZ0)**		
		0	**1**	**Sb**			
	pos	0+1= 1	-1+1 = 0		**Sign bit**		
	0	1	x (0)		RB = a - (not b)		
		1	y (1)		(RB)a = not Sa		
	neg	-1+1 = 0	-2+1 = -1		(RB)b = not Sb		
	1	x (1)	0				
		y (0)	0				
a - b = a + (not b) +1 = (RB)SB - 1 + 1 = (a, notb)							
Deci ar trebui sa caut sa formez produse partiale in forma a - notb							
pentru ca valoarea lor se transfere neschimbata in RB. Fac asta in Booth							

Fig. 3.3 Codare pozitiv negativ - complementat – reprezentare, bit de semn

Bitul de semn în aritmetică redundantă este sintetizat folosind o diagrama Karnaugh evidențiată în figura 3.4.

Figura 3.4 ilustrează diagrama Karnaugh din care s-au sintetizat ecuațiile corespunzătoare semnalelor de intrare pentru cei doi operanzi, X și Y și pentru transporturile asociate acestora. Atât operanzii, cât și transportul sunt reprezentate folosind codarea PNC. Se poate observa din figură că pentru fiecare căsuță din diagrama avem una sau mai multe posibilități de reprezentare. Fiecare căsuță din diagramă are următorul conținut: pe prima linie este semnalul de ieșire, Z, codat PNC, iar pe cea de-a doua linie este transportul de ieșire, C_y, de asemenea codat PNC. Pentru a elimina propagarea transportului de ieșire este necesară crearea dependenței de transportul de intrare doar a unui singur simbol din transportul de ieșire, timp în care celălalt simbol să fie independent de omologul de la intrare. Dacă încercăm să facem simbolul C_{y1} independent, C_{y0} va depinde de C_{in0} și C_{y0} se va propaga. Astfel, pe fiecare coloană C_{y0}trebuie să aibă aceeași valoare de sus până jos.

	Cin1Cin0	-2	-1	0	-1	0	1	0	-1	0	1	2	1	0	1	0	-1
X1X0Y1Y0		0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
-1		-3	-2	-1	-2	-1	0	-1	-2	-1	0	1	0	-1	0	-1	-2
00		00	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab
		00	00	ab 00	00	ab 00	ab	ab 00	00	ab 00	ab	11 ab	ab	ab 00	ab	ab 00	00
0		-2	-1	0	-1	0	1	0	-1	0	1	2	1	0	1	0	-1
01		ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11	ab	00 11
		00	ab 00	ab	ab 00	ab	11 ab	ab	ab 00	ab	11 ab	11	11 ab	ab	11 ab	ab	ab 00
1		-1	0	1	0	1	2	1	0	1	2	3	2	1	2	1	0
11		11 00	ab	11 00	ab	11 00	ab	11 00	ab	11 00	ab	11	ab	11 00	ab	11 00	ab
		00 ab	ab	ab 11	ab	ab 11	11	ab 11	ab	ab 11	11	11	11	ab 11	11	ab 11	ab
0		-2	-1	0	-1	0	1	0	-1	0	1	2	1	0	1	0	-1
10		ab	11 00	ab	11 00	ab	11 00	ab	11 00	ab	11 00	ab	11 00	ab	11 00	ab	11 00
		00	00 ab	ab	00 ab	ab	ab 11	ab	00 ab	ab	ab 11	11	ab 11	ab	ab 11	ab	00 ab

Ca Cy1/0 sa nu se propage e necesar ca macar unul din ei sa depinda numai de intrari iar celalat sa nu depinda de omologul lui de la intrare. Daca incerc sa fac CY1 independent Cy0 va depinde de Cin0 si Cy0 se va propaga. Incerc sa fac Cy0 dependent numai de intrari. Pe fiecare coloana trebuie sa aiba aceesi valoare de sus pana jos:

Fig. 3.4 Diagramă Karnaugh - obținerea ecuațiilor pentru codarea PNC

3.4 ECUAȚII LOGICE PENTRU ADUNARE

În figura 3.10 este evidențiată operația de adunare cu transport unde operanzii, semnalul de transport și rezultatul sunt codați PN. După adunare, se face decodificarea în zecimal folosind formula (3.1). De asemenea, rezultatul este reprezentat și în binar normal.

	1	1	1	0	0	12	plus
	1	0	0	0	0		
			1	1	1	3	
			1	0	0		
Cy	0	1	1	1	1		
	0	0	0	0	1		
S	1	0	0	0	1	minus	
	0	0	0	1	0		
	16	0	0	-2	1		
NB		1	1	1	1		

Fig. 3.5 Exemplificarea unei operații de adunare folosind 2 operanzi codați PN

În urma sintezei PNC, din diagrama Karnaugh s-au obținut următoarele ecuații pentru semnalul de ieșire Z și pentru transport. Aceste ecuații sunt evidențiate în figura 3.6. Se poate observa relația de dependență dintre simbolurile semnalului de ieșire, Z_1 *și* Z_0, conform ecuației (3.9).

Cy0 = (X1 + X0)(Y1 + Y0) = (X1 nor X0) nor (Y1 nor Y0)

Cy1 = (X1 xor X0 xor Y1 xor Y0)Cin0 + (not (X1 xor X0 xor Y1 xor Y0))(X1X0 + Y1Y0)

Z1 = Cin1

Z0 = (X1 xor X0 xor Y1 xor Y0) xor Cin0

Z = Z1 - (notZ0) = Z1 + Z0 + 1

Fig. 3.6 Ecuațiile obținute pentru codarea PNC

În vederea generării produselor parțiale vom implementa algoritmul Booth modificat varianta Radix-16. Acesta constă în divizarea unuia dintre operanzi cu lungimea de 32 biți în grupări de 5 biți. Două grupări succesive vor avea un bit care se va suprapune. Evaluarea acestor biți se face conform tabelului 3.4. Astfel, în funcție de valoarea grupării, celălalt operand se va înmulți cu una din valorile: $0, \pm1, \pm2, \pm4, \pm8\ sau\ \pm5$. Dintre toate acestea înmulțirea cu ±5 este cea mai dificilă, fiind necesar un alt set de ecuații.

		M1(+)	M0(-)	Cy		M1(+)	M0(-)	Cy	
0	0000(0)	0	0	0 (00)	1000(0)	0	8M	0 (00)	16
1	0000(1)	M	0	0 (00)	1000(1)	M	8M	0 (00)	17
2	0001(0)	M	0	0 (00)	1001(0)	M	8M	0 (00)	18
3	0001(1)	2M	0	0 (00)	1001(1)	2M	8M	0 (00)	19
4	0010(0)	4M	2M	0 (00)	1010(0)	2M	8M	0 (00)	20
5	0010(1)	4M	M	0 (00)	1010(1)	notM	4M	+1 (10)	21
6	0011(0)	4M	M	0 (00)	1011(0)	notM	4M	+1 (10)	22
7	0011(1)	4M	0	0 (00)	1011(1)	0	4M	0 (00)	23
8	0100(0)	4M	0	0 (00)	1100(0)	0	4M	0 (00)	24
9	0100(1)	4M	notM	-1 (01)	1100(1)	M	4M	0 (00)	25
10	0101(0)	4M	notM	-1 (01)	1101(0)	M	4M	0 (00)	26
11	0101(1)	8M	2M	0 (00)	1101(1)	0	2M	0 (00)	27
12	0110(0)	8M	2M	0 (00)	1110(0)	0	2M	0 (00)	28
13	0110(1)	8M	M	0 (00)	1110(1)	0	M	0 (00)	29
14	0111(0)	8M	M	0 (00)	1111(0)	0	M	0 (00)	30
15	0111(1)	8M	0	0 (00)	1111(1)	0	0	0 (00)	31

Tabel 3.3 Codare Booth Radix-16 pentru reprezentarea în aritmetică redundantă

În figura 3.11 este evidențiat circuitul modulului generator de produse parțiale reprezentate în aritmetică redundantă. Cele două ieșiri ale acestui circuit reprezintă perechea de biți ce formează produsul parțial. Selecția valorii de ieșire a fiecărui bit din produsul parțial se realizează folosind câte un multiplexor pentru fiecare.

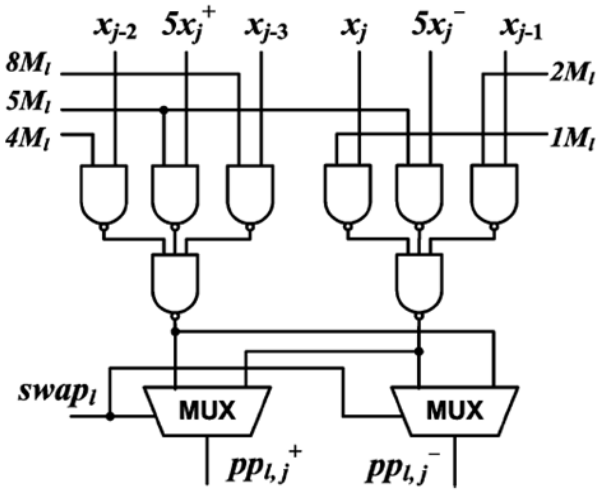

Fig. 3.7 Circuit generator de produse parțiale reprezentate în aritmetică redundantă [15]

În continuare, în vederea obținerii rezultatului înmulțirii, produsele parțiale obținute se vor aduna într-o structură Wallace, ilustrată în figura 3.8. Spre deosebire de cazul adunării produselor parțiale reprezentate în aritmetică normală, aritmetica redundantă ne dă posibilitatea adunării facile, fara a îngreuna adunarea odată cu apropierea de bitul cel mai semnificativ.

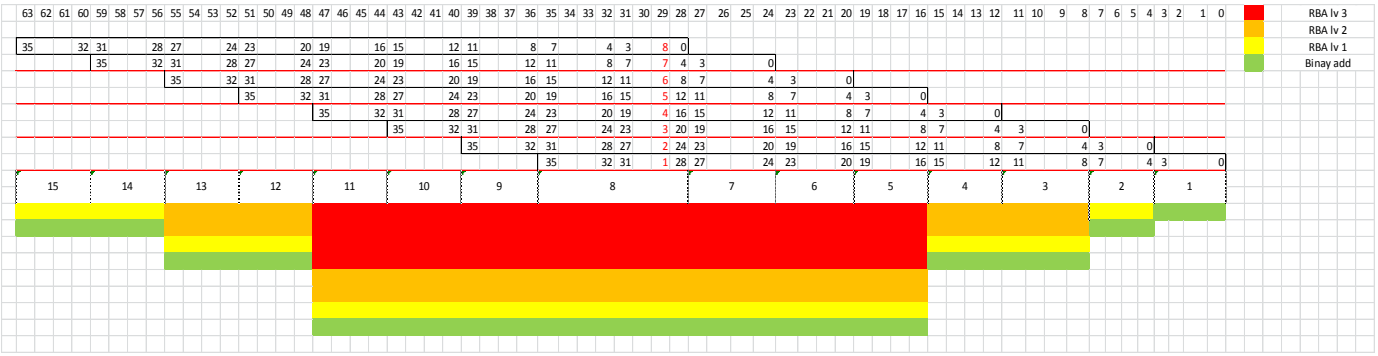

Fig. 3.8 Structura arborelui Wallace

În figură sunt evidențiate cu diferite culori nivelul curent al arborelui Wallace. Cu roșu este primul nivel al arborelui, cu cel mai mare număr de compresoare 4:2, urmat de culoarea portocaliu a nivelului 2, culoarea galben pentru cel de-al treilea nivel și culoarea verde pentru ultimul nivel al arborelui.

Se poate observa că cel mai mare număr de operații de adunare care are loc în interiorul arborelui nu este în apropierea bitului cel mai semnificativ ci la jumătatea distanței dintre bitul cel mai semnificativ și bitul cel mai puțin semnificativ. Acest lucru este datorat lipsei de propagare a semnalului de transport despre care se va discuta în subcapitolul următor.

3.5 COMPRESOR 4:2

Una din metodele de a îmbunătăți viteza unei unități aritmetice este capabilitatea de a aduna numere cu o propagare de transport minimă. De la această premisă a aparut ideea de compresor 4:2. Proprietatea acestuia este că din 4 operanzi la intrare, la ieșire se vor obține 2, adunarea operanzilor și transportului făcându-se separat. Suma operanzilor și transportul se vor aduna la sfârșit în vederea obținerii rezultatului corect.

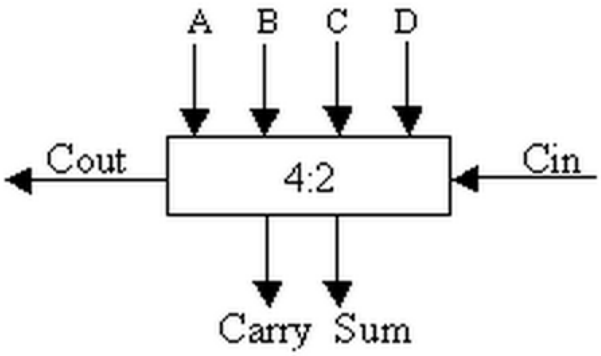

Fig. 3.9 Schema bloc a unui compresor 4:2 [20]

Schema bloc a unui compresor 4:2 este ilustrată în figura 3.7. Caracteristicile acestui bloc sunt:

- ieșirile acestui bloc reprezintă suma celor 5 intrări: 4 semnale de date și un semnal de transport; cu alte cuvinte acesta este un sumator de 5 biți
- pentru a evita propagarea transportului, valoarea transportului de ieșire depinde doar de valorile de intrare nu și de transportul de intrare
- transportul de ieșire reprezintă transportul de intrare pentru compresorul 4:2 de pe nivelul următor

În tabelul 3.3 este evidențiat comportamentul compresorului 4:2 funcție de valorile intrărilor cu ajutorul tabelului de adevăr.

Inputs				Cin = 0		Cin = 1		Cout
A	B	C	D	Carry	Sum	Carry	Sum	
0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0	0
0	0	1	0					
0	1	0	0					
1	0	0	0					
0	0	1	1	0	0	0	1	1
0	1	1	0					
1	1	0	0					
0	1	0	1					
1	0	1	0					
1	0	0	1					
0	1	1	1	0	1	1	0	1
1	1	1	0					
1	1	0	1					
1	1	1	0					
1	1	1	1	1	0	1	1	1

Tabel 3.4 Tabelul de adevăr pentru un compresor 4:2 [20]

Figura 3.8 ilustrează arhitectura unui sumator 4:2 de operanzi cu n biți format din n compresor 4:2.

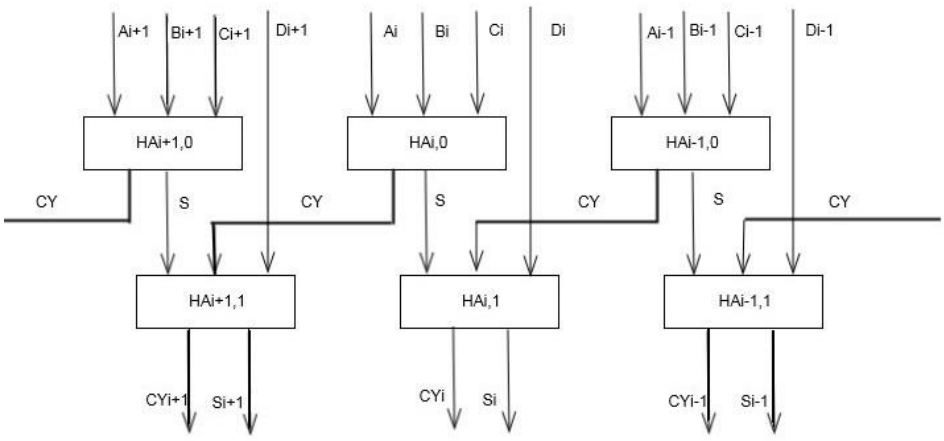

Fig. 3.10 Arhitectura unui bloc format din n compresoare 4:2

În figura 3.7 este reprezentă structura unui sumator Carry Save. Acest tip de sumator se foloseste în cazul realizării unor operații folosind aritmetica redundantă. În figură, semnalele X,T, t și S sunt reprezentate folosing aritmetica redundantă.

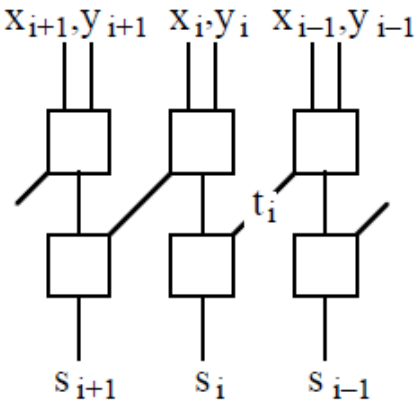

Fig. 3.11 Structura unui sumator Carry Save [15]

Unul din obiectivele principale ale acestui proiect este implementarea unui bloc de înmulțire rapidă cu aritmetică redundantă care să nu propage semnalul de transport la mai mult de două celule consecutive. Avantajul acestei propagări limitate este protecția la erori. În cazul în care apare o eroare într-unul din blocuri aceasta nu se va mai propaga și nu va altera întreg rezultatul ci doar o parte din acesta.

Acest tip de propagare al transportului în maxim două celule consecutive se poate obține folosind sumatoare Carry-Save. Modul de funcționare este evidențiat în figura 3.10. În cazul reprezentării în aritmetica redundantă, semnalul de transport este compus din doi biți. Se observă condiționarea unuia din biții semnalului de transport de a se include în rezultatul celulei curente. Cel de-al doilea bit al semnalului de transport se va propaga la celula următoare reluând procesul.

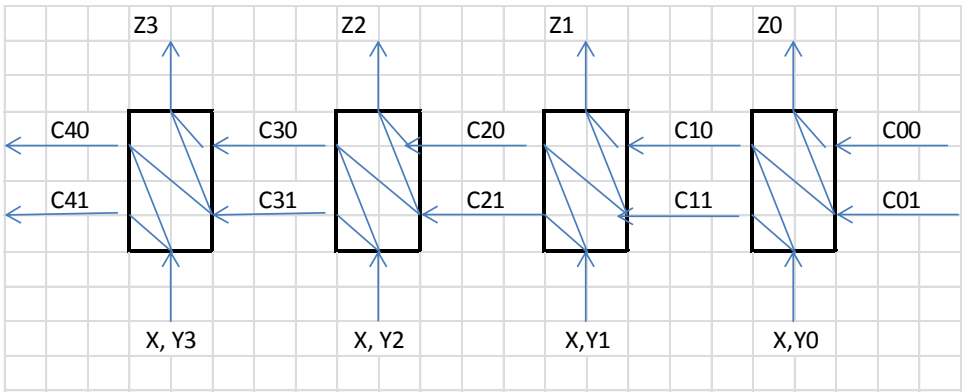

Fig. 3.12 Structură de propagare a semnalului de transport

3.6 FUNCȚIONAREA BLOCULUI DE ÎNMULȚIRE FOLOSIND ARITMETICĂ NORMALĂ

3.6.1 DESCRIEREA FUNCȚIONĂRII

Blocul de înmulțire cu aritmetică normală primește doi operanzi cu o lungime 8 biți și întoarce un rezultat cu o lungime 16 biți. Modulul de top nb_multiplicator funcționează sincron cu un semnal de ceas, clk, cu frecvența de 50MHz. De asemenea, modulul prevede și un semnal de reset, por, activ în 1 logic, acesta fiind asincron cu semnalul de ceas.

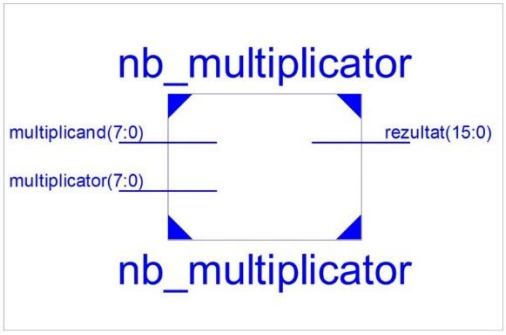

Fig. 3.13 Modul top – bloc de înmulțire cu aritmetică normală

Modulul ierarhic superior este format din 3 module principale, fiecare având o funcție specifică:

- top_booth
- pp_gen
- top_sum

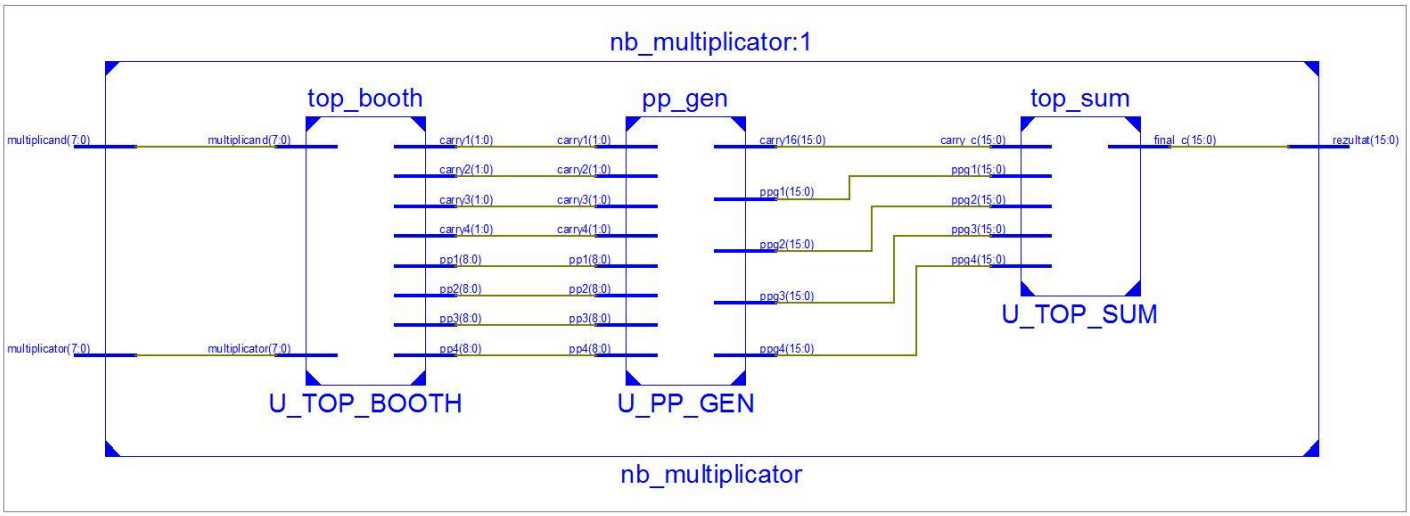

Fig. 3.14 Modul top – blocuri componente

Pentru implementarea unui bloc de înmulțire rapid, folosirea algoritmului Booth modificat este o metodă eficientă de reducere a numărului de produse parțiale, formând grupuri de biți

consecutivi în unul din cei doi operanzi. Operandul căruia i se formează grupuri de biți folosind algoritmul Booth modificat se numește deînmulțit, iar celălalt operand se numește înmulțitor.

Modulul top_booth este format din 4 module Booth care au funcția de a returna câte un produs parțial de 9 biți. Înmulțitorul este împărțit în grupuri de câte 3 biți. Astfel, conform algoritmului Booth Radix-4, ilustrat în figura 3.14 se va obține produsul parțial realizând înmulțirea deînmulțitului cu una din valorile 0,1, 2,-2 sau -1. [19]

Modified (Radix 4) Booth				
mpl_i+1	**mpl_i**	**mpl_i-1**	**PP**	**Obs**
0	0	0	x 0	no string
0	0	1	x 1	end of string
0	1	0	x 1	single 1
0	1	1	x 2	end of string
1	0	0	x (-2)	start of string
1	0	1	x (-1)	start of string and end of string
1	1	0	x (-1)	start of string
1	1	1	x 0	middle of string

Tabel 3.5 Algoritm Booth modificat (Radix 4)

Modulul top_booth are două semnale de intrare de 8 biți, cei doi operanzi, și opt semnale de ieșire: patru semnale corespunzătoare produselor parțiale și patru semnale reprezentând transportul, cu o lungime de 2 biți, corespunzător fiecărui produs parțial generat.

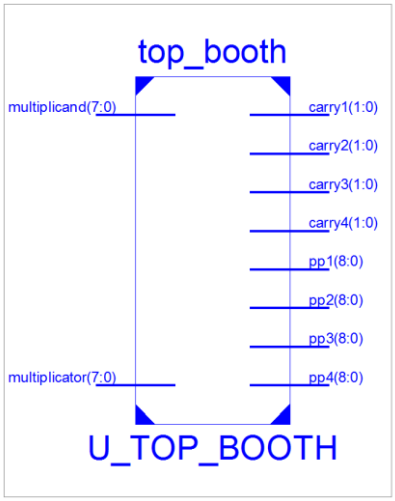

Fig. 3.15 Bloc top_booth

Având în vedere că lungimea operanzilor este de 8 biți, se vor forma, conform algoritmului lui Booth modificat, 4 grupuri de câte 3 biți. Folosind aceste grupuri de 3 biți se vor genera produsele parțiale, realizându-se operația de înmulțire a deînmulțitului cu numărul din tabel corespunzător grupului de 3 biți. Astfel, se vor genera 4 produse parțiale, fiecare produs parțial fiind întors de câte un submodul booth. Acest lucru se poate observa în figura 3.16.

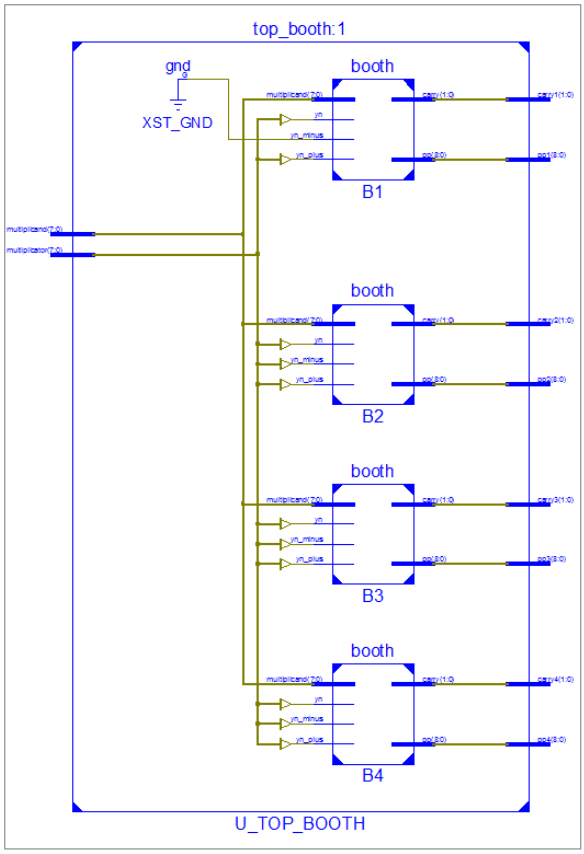

Fig. 3.16 Bloc top_booth –blocuri componente

Modulul pp_gen realizează alinierea produselor parțiale. Astfel se shiftează la stânga fiecare produs parțial conform numărului de ordine. De asemenea, se face extensia bitului de semn.

În cazul în care s-a facut o înmulțire cu -1 sau -2, în fiecare modul Booth se generează un semnal de carry de 2 biți, urmând ca în modulul Partial_product_generator să se facă concatenarea celor patru semnale de carry.

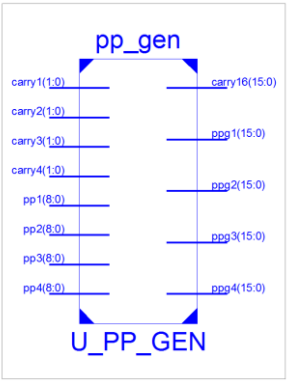

Fig. 3.17 Bloc pp_gen

Modulul pp_gen are opt semnale de intrare, primite de la modulul top_booth, acestea fiind patru semnale corespunzătoare produselor parțiale și patru semnale reprezentând transportul corespunzător fiecărui produs parțial generat. La ieșire avem patru semnale reprezentând produsele parțiale aliniate conform numărului de oridine și cu extensie a bitului de semn și un semnal ce conține toate semnale de transport generate anterior de modulele booth, concatenate și aliniate corespunzător numărului de ordine.

Modulul top_sum este un arbore de sumatoare, acest modul având cele cinci semnale de intrare provenite de la modulul pp_gen și un semnal de ieșire reprezentând rezultatul înmulțirii.

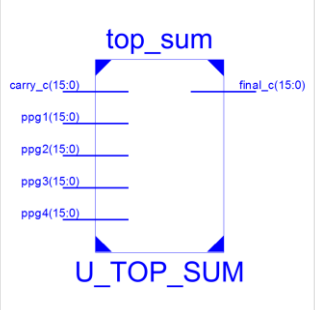

Fig. 3.18 Bloc top_sum

Arborele de sumatoare conține pe primul nivel produsele parțiale provenite de la modulul pp_gen, care se însumeaza două câte două. Pe cel de-al doilea nivel rezultatele obținute din primul nivel, două semnale de 16 biți, se însumează. Pe cel de-al treilea nivel se însumează rezultatul obținut anterior cu semnalul de carry.

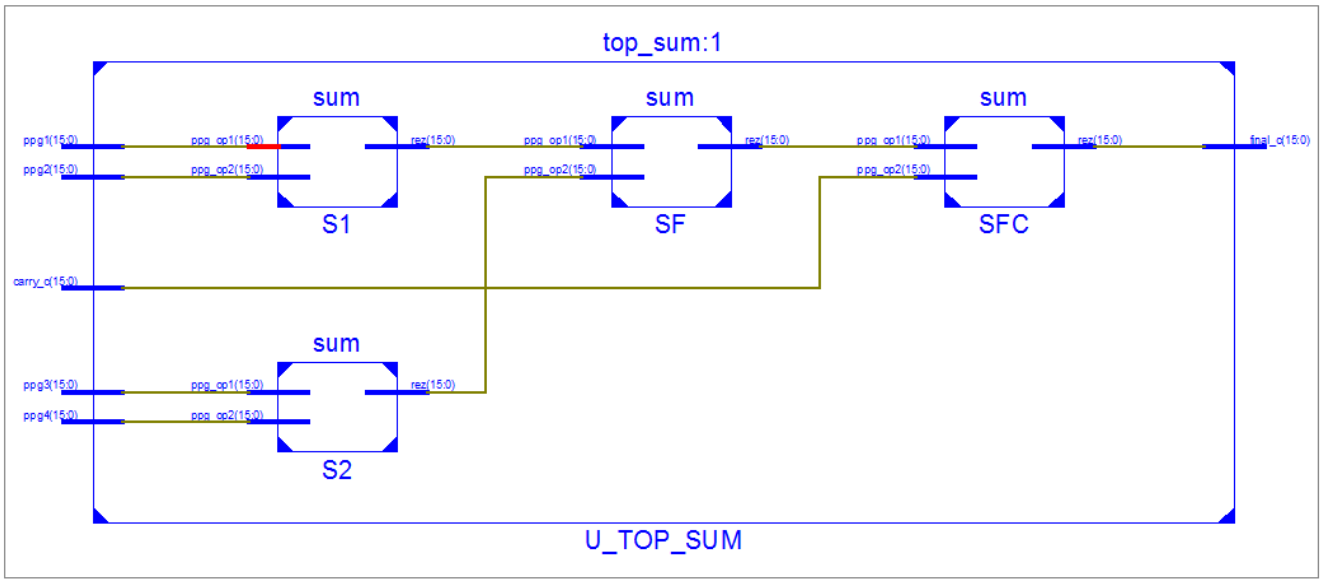

Fig. 3.19 Bloc top_sum – blocuri componente

3.6.2 REZULTATELE SIMULĂRII

Am realizat un modul de testare în care verific rezultatul multiplicatorului pentru toate valorile posibile, cu semn, pentru cei doi operanzi. Operanzii iau valori în intervalul [-128,127]. Pentru verificarea corectitudinii rezultatului obținut din blocul de înmulțire, operația se realizeaza în doua moduri: înmulțind valorile date operanzilor folosind semnul „*” și folosind modulul implementat.

Cele două rezultate obținute se compară, iar rezultatul comparației este ilustrat prin evoluția semnalului error_tb. Semnalul error_tb ia valoarea „0” în momentul în care cele doua valori sunt egale și valoarea „1” când cele două valori nu corespund.

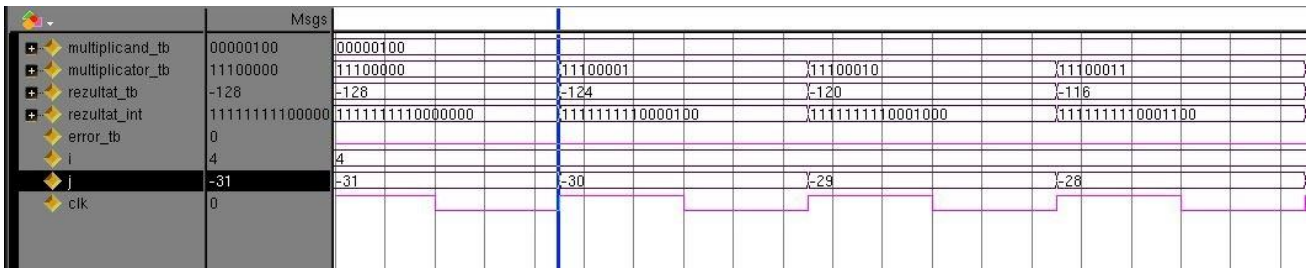

Fig. 3.20 Simulare bloc înmulțire cu aritmetică normală

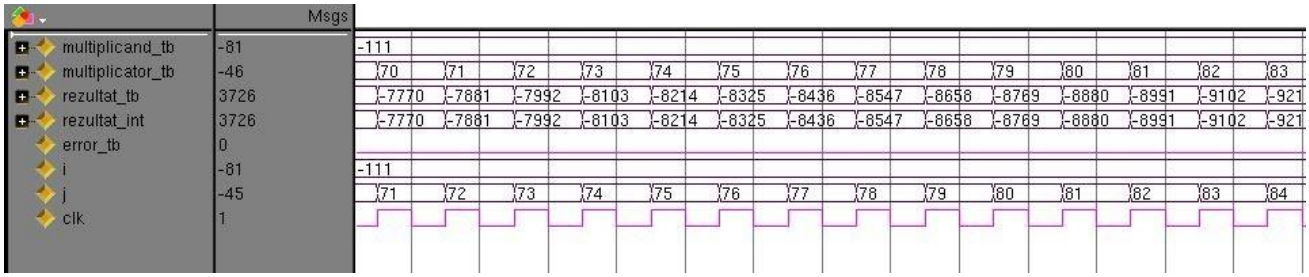

Fig. 3.21 Simulare bloc înmulțire cu aritmetică normală

Se observă că semnalul error_tb are numai valoarea 0, ceea ce sugerează buna funcționare a modulului implementat.

3.7 FUNCȚIONAREA BLOCULUI DE ÎNMULȚIRE FOLOSIND ARITMETICĂ REDUNDANTĂ

3.7.1 DESCRIEREA FUNCȚIONĂRII

Multiplicatorul digital este o unitate aritmetică cheie folosită în microprocesoare, procesoare digitale de semnal, filtre digitale etc. De-a lungul timpului a apărut necesitatea unor multiplicatoare rapide, capabile să livreze rezultatul după o perioadă de ceas de la primirea operanzilor.

Utilizarea reprezentării numerelor folosind aritmetica redundantă a fost introdusă de Avizienis în vederea aplicării în aritmetica paralelă rapidă. Acest concept a fost utilizat de Takagi et al. pentru multiplicarea rapidă.

Similar blocului de înmulțire cu aritmetică normală, un bloc de înmulțire cu aritmetică redundantă este divizat în trei etape și este alcătuit din patru module principale: codor Booth, generator de produse parțiale cu reprezentare în aritmetică redundantă, acumulator de produse pațiale cu reprezentare în aritmetică redundantă și bloc de conversie din reprezentare în aritmetică redundantă în reprezentare în aritmetică normală. Scopul principal al ultimului bloc este de a comunica cu perifericele care, în general, lucrează cu un sistem de numerație convențional.

Câteva avantaje ale folosirii unui multiplicator rapid realizat folosind aritmetica redundantă sunt:

- Obținerea rezultatului înmulțitii după un singur tact de ceas de la primirea operanzilor

- Adunare fără propagare de carry, înlocuind astfel adunarea numerelor în complement față de 2
- Reducerea înălțimii arborelui Wallace – necesitatea unui număr mai mic de sumatoare față de cazul multiplicatorului realizat utilizând aritmetica normală

Prezenta lucrare se concentrează pe utilizarea operanzilor cu o lungime putere a lui 2, mai exact, 32 biți, în vederea exploatării reducerii numărului de produse parțiale al arborelui de însumare folosit în interiorul sumatorului cu aritmetică redundantă.

Arhitectura unui bloc de înmulțire cu aritmetică redundantă este ilustrata în figura 3.21. Modulul de top, ierarhic superior, mply_top are două semnale de intrare, cu o lungime de 32 de biți, reprezentând operanzii. Semnalul de ieșire, cu o lungime de 64 biți, reprezintă rezultatul înmulțirii. Modulul mply_top funcționează sincron cu un semnal de ceas, clk, cu frecvența de 50MHz. De asemenea, modulul prevede și un semnal de reset, por, activ în 1 logic, acesta fiind asincron cu semnalul de ceas.

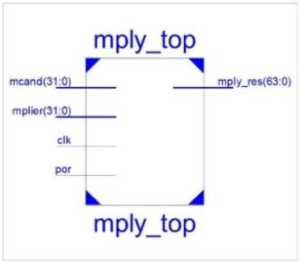

Fig. 3.22 Modul top – bloc de înmulțire cu aritmetică redundantă

În figura este detaliată schema RTL a modulului mply_top. În componența aceseia intră 2 module principale: pp_gen și wallace.

Se pot observa în schema și existența a trei registre, cele două registre de la intrare fiind folosite pentru preluarea operanzilor, înaintea prelucrării, iar cel de-al treilea registru fiind folosit pentru a încărca rezultaul livrat de modul, spre a fi trimis către ieșire.

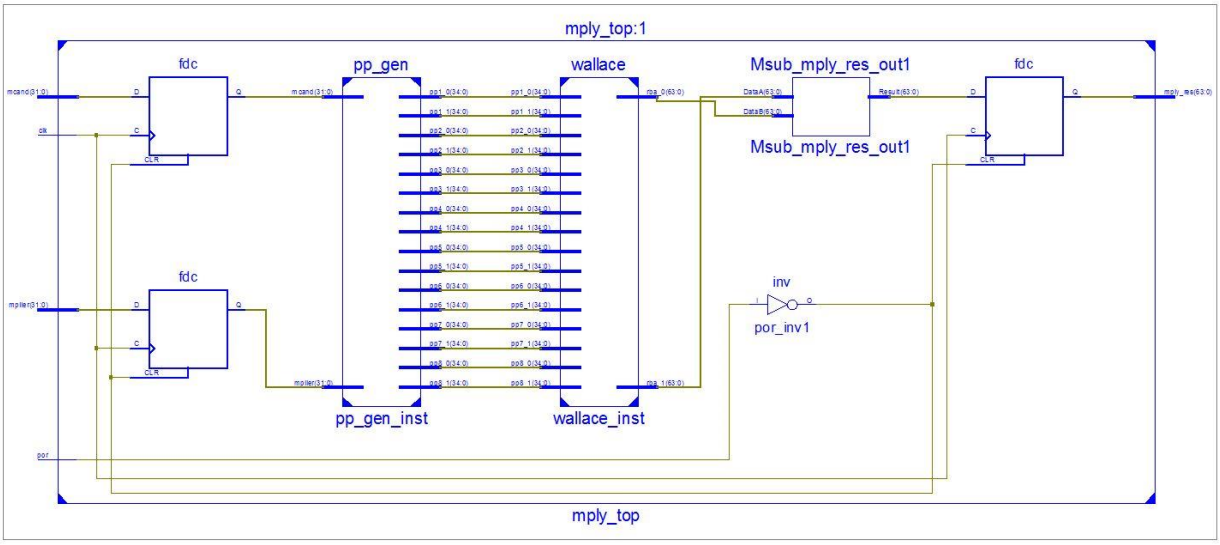

Fig. 3.23 Modul top – Blocuri componente

Pentru implementarea unui bloc de înmulțire rapid, folosirea algoritmului Booth modificat este o metodă eficientă de reducere a numărului de produse parțiale.

Modulul pp_gen prezintă două semnale de intrare, cei doi operanzi, cu o lungime de 32 de biți. Operanzii sunt reprezentați folosind aritmetica redundantă. La ieșirea acestui bloc vom obține produsele parțiale ce urmează să fie folosite pentru obținerea rezultatului înmulțirii. În total avem opt produse parțiale. În figura 3.23 observăm 16 semnale de ieșire, fiecare având o lungime de 35 biți. Cu ajutorul acestor semnale se vor forma perechi, fiind, de fapt, reprezentarea în aritmetică redundantă a produselor parțiale.

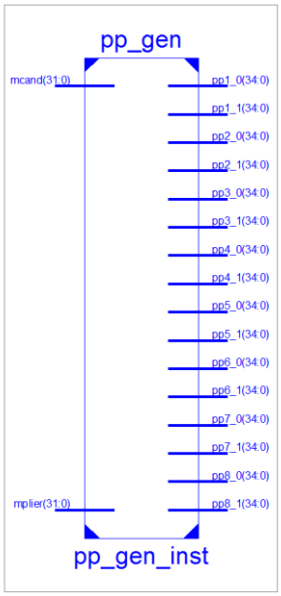

Fig. 3.24 Modul pp_gen

Modulul pp_gen are nouă blocuri componente: opt blocuri one_pair_pp și un bloc x5. În figura 3.24 se pot observa legaturile dintre aceste blocuri.

Așa cum a fost explicat în subcapitolul 3.5, produsele parțiale sunt generate utilizând algoritmul lui Booth modificat Radix-16. Conform acestui algoritm, din unul dintre cei doi operanzi se vor forma grupuri a câte 5 biți, urmând ca, în funcție de valoarea acestora, să se efectueze o înmulțire a celuilalt operand cu : $0, \pm 1, \pm 2, \pm 4, \pm 8$ sau ± 5. Dintre acestea, înmulțirea care poate provoca probleme este cea cu 5 de aceea. Din aceste considerente înmulțirea cu 5 se va trata separat în submodulul x5.

Atât componenta x5 cât și cele 8 componente one_pair_pp întorc rezultate reprezentate în aritmetică redundantă, având câte 2 semnale de ieșire identice ca lungime.

Submodulul x5 are două semnale de intrare, înmulțitorul și deînmulțitul, cu o lungime de 32 de biți și un semnal de ieșire, de o lungime de 35 biți. Acest submodul are funcția de a realiza înmulțirea cu 5 a deînmulțitului, în cazul în care, dupa evaluarea valorii uneia din perechiile de 5 biți formate din înmulțitor corespunde unei înmulțiri cu 5. Cele patru combinații posibile de astfel de perechi sunt următorele, conform tabelului 3.3: 01001, 01010, 10101 și 10110.

Blocul de înmulțire implementat în prezenta lucrare prevede doi operanzi cu o lungime de 32 de biți. În consecința, se vor forma 8 grupuri de câte 5 biți cu ajutorul cărora se vor genera

produsele parțiale. Grupurile se vor suprapune cu câte un bit, cu alte cuvinte, două grupuri consecutive vor avea câte un bit comun. În cazul primei grupări, este necesară o concatenare la dreapta a pimilor 4 biți mai puțin semnificativi cu „0”.

Submodulul one_pair_pp are 4 semnale de intrare și 2 semnale de ieșire. La intrare vom avea un semnal de 32 de biți corespunzător deînmulțitului, un semnal de 5 biți corespunzător grupării de 5 biți formate din înmulțitor și 2 semnale reprezentând semnalele de ieșire din submodulul x5. Semnalele de ieșire din submodulul one_pair_pp reprezintă produsul parțial reprezentat în aritmetică redundantă corespunzător gruparii de 5 biți.

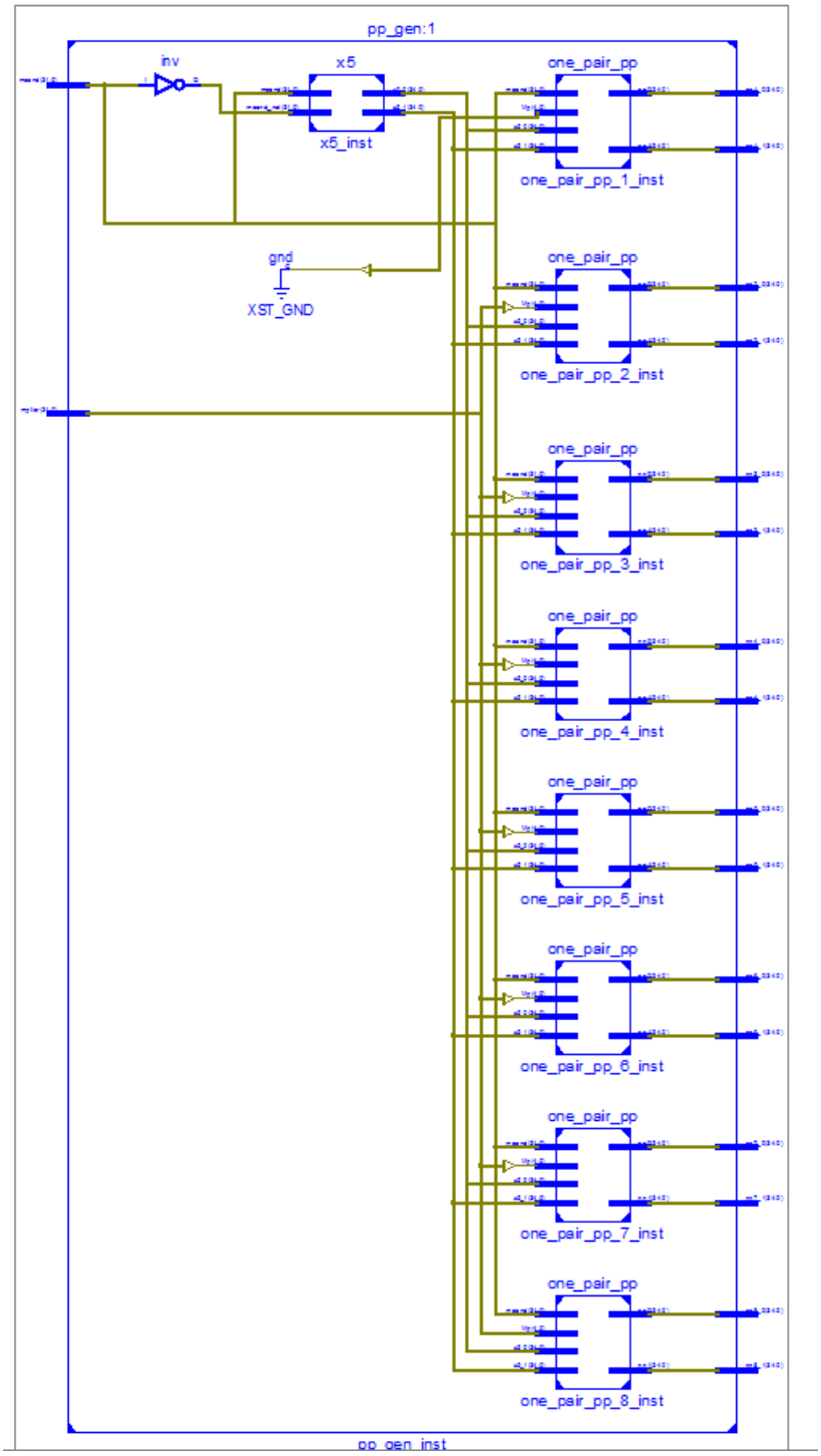

Fig. 3.25 Modul pp_gen - blocuri componente

Fiecare submodul one_pair_pp conține componenta booth, ilustrată în figura 3.25. Rolul acestei componente este de a evalua grupul de 5 biți corespunzător numărului de ordine al componentei one_pair_pp.

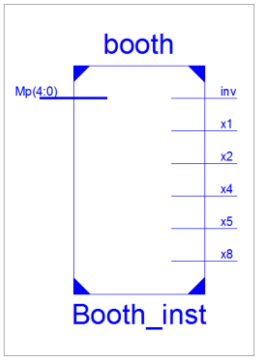

Fig. 3.26 Componentă booth

Cel de-al doilea modul principal al blocului de înmulțire este blocul wallace, ilustrat în figura 3.26. Semnalele de intrare sunt cele 8 produse parțiale reprezentate în aritmetică redundantă generate de modulul pp_gen. Semnalele de ieșire ale acestui modul este rezultatul înmulțirii celor doi operanzi, de asemenea reprezentat în aritmetică redundantă.

În esență, modulul wallace reprezintă un arbore de însumare Wallace, cunoscut sub numele de Wallace Tree. În componența acestui modul intră submodulul RBA_cell_PN, instanțiat iterativ. Componenta RBA_cell_PN reprezintă un compresor 4:2 despre care s-a discutat în subcapitolul 3.5. Pe primul nivel al arborelui Wallace vom avea 4 compresoare 4:2, semnalele de intrare ale acestora fiind produsele parțiale generate de modulul pp_gen grupate 2 câte 2. Pe cel de-al doilea nivel al arborelui de însumare vom avea 2 compresoare 4:2 ale cărui intrări sunt rezultatele obținute de la nivelul anterior. Pe cel de-al treilea nivel se vor însuma rezultatele obținute din nivelul anterior, tot printr-un compresor 4:2. Rezultatul obținut din nivelul de ordin 3 reprezintă rezultatul înmulțirii celor 2 operanzi, codat în aritmetică redundantă.

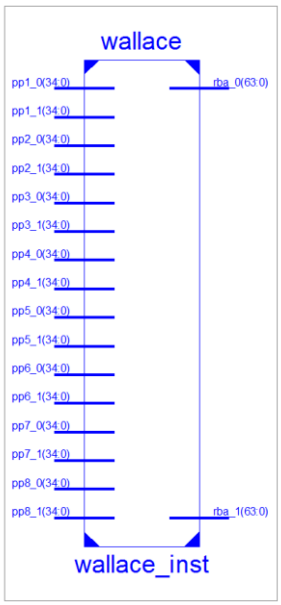

Fig. 3.27 Modul wallace

3.7.2 REZULTATELE SIMULĂRII

Rezultatele simulării blocului de înmulțire cu aritmetică redundantă sunt evidențiate în figura 3.27. Ținând cont că modulul este sincron se vor urmări evenimentele semnalelor în momentul apariției frontului crescător al ceasului.

Fig. 3.28 Rezultatele simulării blocului de înmulțire cu aritmetică redundantă

Primul cursor marchează încărcarea operanzilor în registre pe frontul crescător al ceasului. Cel de-al doilea cursor evidențiază rezultatul înmulțirii obținut după o perioadă de ceas.

4 PROIECTARE ȘI IMPLEMENTARE BLOC RAPID DE ÎNMULȚIRE UTILIZÂND VIRTEX-7 FPGA

4.1 DESCRIEREA FUNCȚIONĂRII

În prezenta lucrare s-a evidențiat necesitatea implementării unui bloc de înmulțire rapidă ce utilizează aritmetica redundantă. Pe lângă acest bloc, ce presupune logica înmulțitorului, pentru a demonstra funcționarea acestui bloc pe placa de dezvoltare VC707 a fost necesară implementarea mai mai multor module. Modulul principal, top_multiplicator, ierarhic superior este prezentat în figura 4.1 și integreaza cele cinci blocuri principale ce asigură buna funcționare a întreg modulului.

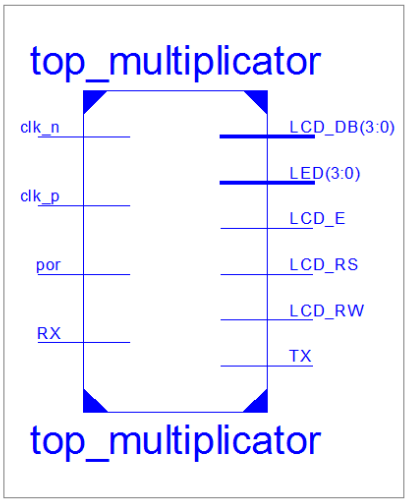

Fig. 4.1 Modul principal - bloc de înmulțire rapidă

Pentru înțelegerea modului de funcționare a blocului de înmulțire proiectat este necesară descrierea în detaliu a etapelor parcurse. Pentru a realiza o înmulțire sunt necesari doi operanzi. Acești operanzi sunt specificați de utilizator, folosind tastatura calculatorului la care este conectată placa de dezvoltare VC707. Operanzii sunt trimiși către modulul implementat utilizând transmisia serială, întrucât ne este oferită posibilitatea de a stabili conexiunea între calculator și FPGA folosind o punte USB-to-UART. Deoarece se folosește o transmisie serială, este necesară o conversie serial-paralelă, urmată de decodificarea cifrelor primite de la tastatură, din cod ASCII în reprezentare în bază 16, hexazecimal. După ce operanzii au fost decodificați și încărcați în registrele corespunzătoare, se va realiza înmulțirea propriu-zisa. Rezultatul, livrat după o perioadă de ceas, urmează sa fie preluat de un modul cu funcția de codificare din hexazecimal în cod ASCII. După realizarea codificării, fiecare niblu din rezultat, sub formă de octet corespunzător unui caracter în ASCII este încărcat într-un bloc de memorie cu proprietatea unui FIFO (First In First Out). Din acest bloc de memorie se vor prelua date, octet cu octet, de un modul destinat să comunice cu afișajul disponibil pe placa. Acești octeți sunt preluați unul câte unul și afișați pe LCD. De asemenea, imediat după conversia datelor primite de la tastatură, acestea se vor trimite către modulul de afișare, în vederea vizibilității operanzilor.

Cele cinci submodule ce intră în componența blocului de înmulțire sunt următoarele:

1. Blocuri necesare pentru asigurarea semnalului de ceas - modulele IBUFGDS + freq_div
2. Protocol de comunicație USB-to-UARt - modulul uart_fpga
3. Bloc de înmulțire cu aritmetică redundantă - modulul rb_multiplicator
4. Memorie de stocare a datelor - modulul fifo
5. Protocol de comunicație cu afișajul - modulul lcd_init

Legăturile dintre aceste module sunt evidențiate în figura 4.2, iar detalierea funcțiilor îndeplinite de acestea va fi explicată în continuare.

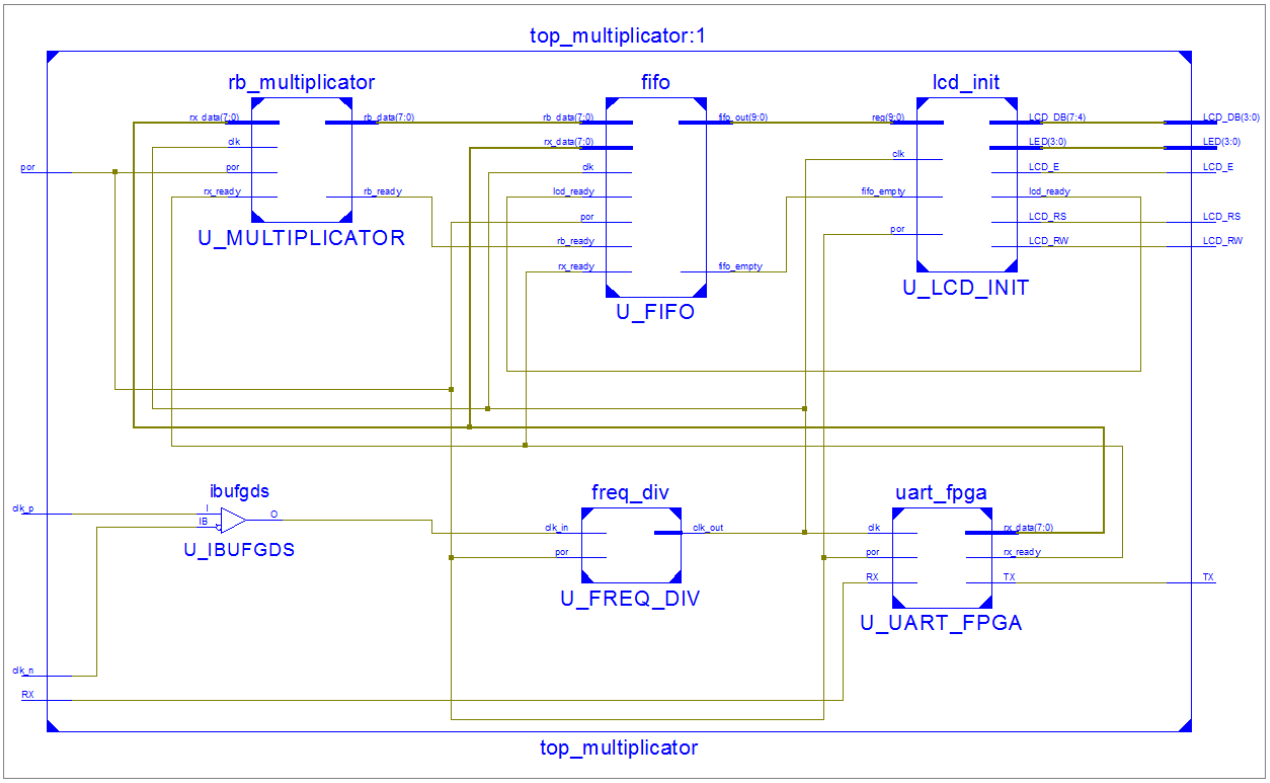

Fig. 4.2 Modul principal – blocuri componente

Conexiunile logice dintre fiecare bit al porturilor de intrare și ieșire ale modulului principal, top_multiplicator, și pinii cipului FPGA Virtex-7 fac posibilă conectarea dispozitivelor externe necesare de pe placa de dezvoltare VC707 la modulul VHDL definit. Aceste legături nu vor fi permanente, astfel, la fiecare configurare a plăcii se vor specifica programului de sinteză, înaintea etapei de implementare. Specificarea legăturilor se dau într-un fișier de constrângeri, cu extensia .ucf, inclus în proiectul din ISE Design Suite.

4.1.1 BLOCURI NECESARE PENTRU ASIGURAREA SEMNALULUI DE CEAS

Semnalul de ceas provenit de la cipul FPGA este o pereche de semnale diferențiale. În acest caz, apare necesitatea utilizării unei primitive IO diferențiale. Aceasta primitivă se numește IBUFGDS și este un modul cu doi pini de intrare, corespunzatori celor două semnale de ceas, N și P, provenite de la FPGA în pereche diferențială, și un semnal de ieșire. Acest tip de primitvă este aleasă conform standardului corespunzător semnalelor folosite. Primitiva IBUFGDS este specifică

semnalelor de ceas în pereche diferențială puse la dispoziție de cipul FPGA Virtex-7 și este ilustrată în figura 4.3. [17]

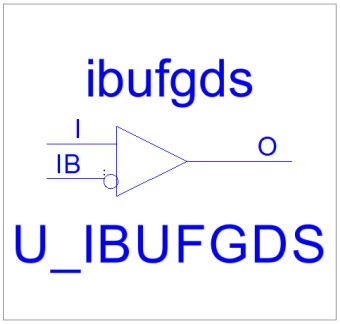

Fig. 4.3 Componentă IBUFGDS

Semnalul de ceas obținut lucrează la o frecvență de 200 MHz. Din cauza unor restricții hardware impuse de transmisia serială și afișajul LCD, a apărut necesitatea divizării frecvenței ceasului de la frecvența inițială de 200 MHz la o frecvență de 50 MHz. Acest lucru a fost realizat cu un submodul de divizare de frecvență, ilustat în figura 4.4.

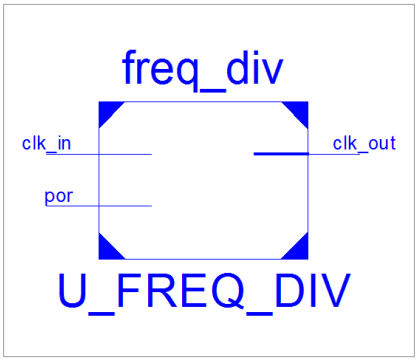

Fig. 4.4 Componentă freq_div

Divizorul de frecvență are funcția de a diviza frecvența inițială de 4 ori. În esență, modulul de divizare de frecvență este un numărător. Există o variabilă internă incrementată ce are o lungime de 2 biți, la care se adaugă 1 odata cu frontul crescător al semnalului de intrare, în acest caz, semnalul de ceas provenit de la FPGA. Semnalul de la ieșirea modulului de divizare este bitul cel mai semnificativ al acestei variabile.

4.1.2 PROTOCOL DE COMUNICAȚIE USB-TO-UART

Un receptor/transmiţător asincron universal (UART) este un modul hardware implementat în FPGA care transformă datele din serial în paralel şi viceversa. Faptul că acest modul este universal indică faptul că formatul datelor şi a vitezelor de transmisie este unul configurabil.

4.1.2.1 Generalități

Modulul UART primeşte octeţi de date şi îi transmite serial, secvenţial. Partea de recepţie ia biţii de date care vin serial şi îi reîmpachetează transmiţându-i mai departe paralel. Fiecare modul UART conţine un registru de shiftare, fundamentul oricărei conversii paralel-seriale. Motivul pentru care se utilizează transmisia serială este destul de evident, folosirea unui număr redus de conexiuni

fizice. Comunicaţia în protocol poate fi simplex (într-o singură direcţie), half-duplex (într-o singură direcţie la un moment de timp dat, dispozitivele trimit şi recepţionează date pe rând) şi full-duplex (amândouă dispozitivele pot trimite şi recepţiona date la acelaşi moment de timp). În cazul de faţă s-a optat pentru un transfer full-duplex.

În starea de inactivitate linia de comunicaţie este ţinută în ’1’ logic. Un pachet (frame) de date este alcătuit din: bitul de start care este întotdeauna ’0’ şi care indică faptul că un nou caracter trebuie recepţionat, urmează apoi biţii de date propriu-zişi care pot fi în număr de 5 până la 9 şi eventual un bit de paritate după care urmează bitul (biţii) de stop care este întotdeauna ’1’ logic şi care indică terminarea transmisiei şi care poate fi folosit în resincronizare şi pentru semnalarea eventualelor erori de nepotrivire a vitezei. S-a optat pentru un modul uart simplu cu bit de start, 8 biţi de date, fără bit de paritate şi un singur bit de stop.

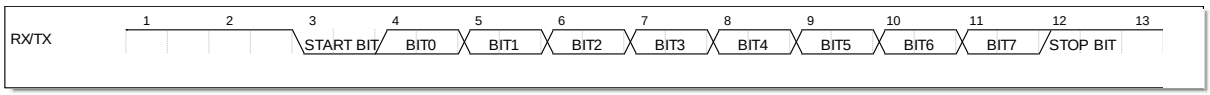

Fig. 4.5 Comunicaţia UART, exemplu pachet de date valid [18]

Întregul modul UART trebuie să funcţioneze pe o frecvenţă de cel puţin 8 ori mai mare decât frecvenţa transmisiei de date pentru a asigura un grad de încredere a recepţionării datelor cât mai mare. Receptorul verifică starea semnalului recepţionat, RX, la fiecare semnal de ceas. Atunci când este detectat bitul de start şi acesta este valid cel puţin jumătate de perioadă de bit atunci acesta este considerat un bit de start valid semnalând începerea comunicaţiei. Urmează apoi recepţionarea biţilor de date cărora li se face eşantionare exact la mijlocul perioadei de bit alocate lor pentru a reduce posibilitatea de eşantionare eronată. La sfârşit se verifică faptul că bitul de stop este într-adevăr ’1’ şi se validează frame-ul recepţionat setând un indicator. Acest indicator poate fi folosit ca întrerupere în sistemele care au implementate astfel de facilităţi. Se pot folosi de asemenea memorii tampon de date gen FIFO, urmând ca sistemul de prelucrare să ia datele la un moment ulterior.

La partea de transmisie lucrurile stau mai simplu deoarece controlul aparţine modului în cauză. Când data este pusă în registrul de shiftare modulul trebuie să trimită mai întâi bitul de start apoi cei 8 biţi de date începând cu bitul 0 până la bitul 7 şi apoi bitul de stop. De asemenea este util de semnalat în exterior când modulul de transmisie este în cursul unei operaţii deoarece vitezele de comunicaţie a UART-ului sunt de obicei mult mai mici decât vitezele de funcţionare a procesoarelor sau modulelor ce le folosesc.

4.1.2.2 Modul baud-rate generator

Acest modul are rolul de a genera semnalul care să indice fie momentul când trebuie să se facă eşantionare, pentru receptor, fie momentul când trebuie să se modifice semnalul de transmisie, pentru transmiţător.

Modulul a fost gândit cu semnal de enable pentru ca să nu funcţionează când nu există activitate pe UART. De asemenea, modulul este configurabil atât ca şi durată a ciclului de numărare (durata perioadei de bit în ciclii de ceas al sistemului) cât şi ca momentul când trebuie generat semnalul de eşantionare/modificare.

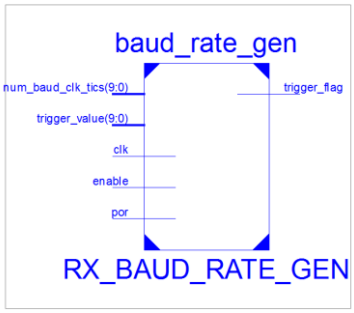

Fig. 4.6 Modul baud_rate_gen - porturi

Această entitate are în componenţă un numărător cu semnal de activare configurabil atât ca şi număr de biţi cât şi ca valoare de reset. Semnalul num_baud_clk_tics indică valoare de reset a numărătorului, iar semnalul trigger_val indică valoare la care semnalul trigger_flag are valoare ’1’ logic.

Pentru viteza de lucru de 9600 bauds la care lucrează sistemul semnalul num_baud_clk_tics trebuie setat la 868 (1/9600 = 104166 ns perioadă de bit, iar semnalul de ceas are o perioadă de 20 ns).

4.1.2.3 Receptorul UART: RX

Receptorul UART are în componenţă modulul baud_rate_gen despre care s-a vorbit mai sus. Pentru acesta am setat parametri după cum urmează:

Parametru	Valoare
num_baud_clk_tics	5208
trigger_val	2604

Tabel 4.1 Parametri baud_rate_gen UART RX

Modulul stă într-o stare de inactivitate, idle, atât timp cât linia de RX este în ’1’ logic. Funcţionarea propriu-zisă începe atunci când se produce o tranziţie pe RX din ’1’ în ’0’. Atunci circuitul trece într-o stare de start în care stă jumătate de perioadă de bit, 4340ns, la sfârşitul căreia se verifică din nou dacă linia RX este în ’0’. Dacă această condiţie este îndeplinită atunci circuitul trece în stările de recepţie propriu-zisă de date de date, începând cu starea bit0, la sfârşitul căreia se va eşantiona bitul 0 dintre cei opt biţi ce urmează a fi recepţionaţi. Acest lucru se realizează folosind un registru de deserializare.

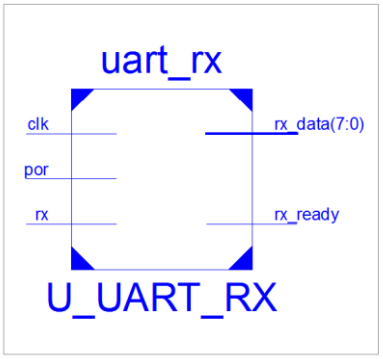

Fig. 4.7 UART RX - porturi

Maşina cu stări finite care stă la baza funcţionării receptorului este redată în figura 4.8. În fiecare stare în afară de stările idle start şi stop, în funcţionarea normală, se aşteaptă 8680 ns înainte de a face trecerea într-o nouă stare odată cu apariţia semnalului trigger_flag generat de modulul baud_rate_gen. Acest modul funcţionează în toate stările exceptând starea de idle.

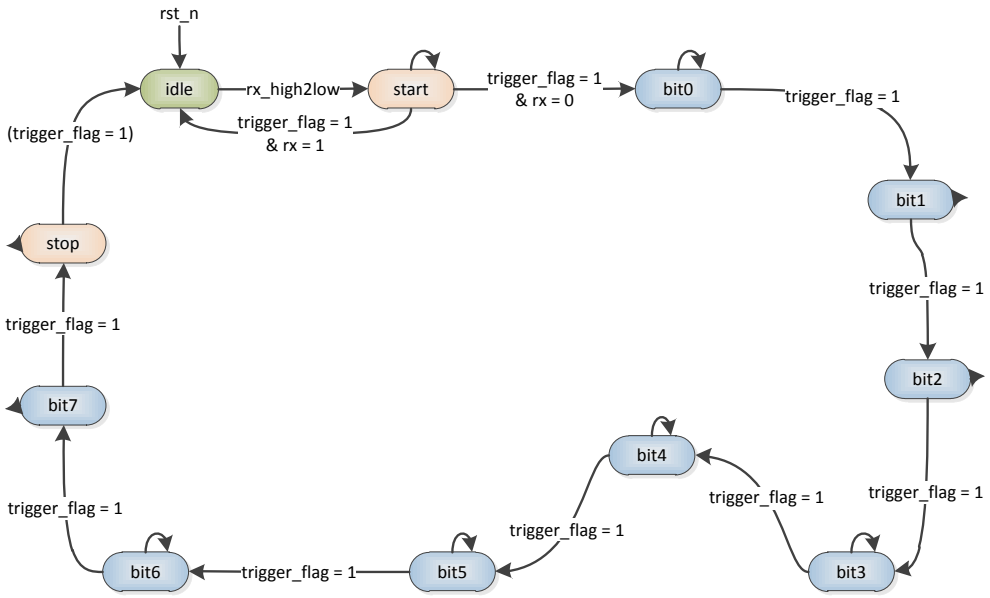

Fig. 4.8 Automat de stări pentru receptorul UART

Un caracter se consideră că este recepţionat cu succes dacă atât bitul de start este ’0’ cât şi bitul se stop este ’1’. Dacă aceste două condiţii nu sunt îndeplinite circuitul se reîntoarce în starea de idle fără însă a semnala în exterior recepţionarea cu succes a datelor prin asertarea semnalului ready, activ în ’1’.

Detecţia tranziţiei din ’1’ în ’0’ a lui RX la începutul recepţionării frame-ului este obţinută prin folosirea a două bistabile cascadate pe semnalul în cauză, pe lângă cel de sincronizare. Când ieşirile celor două bistabile sunt în ’0’, pentru bistabilul ce apare primul în lanţ, şi ’1’ pentru al 2-lea atunci există o tranziţie dorită, codată în figura 4.8 sub numele de rx_high2low.

4.1.2.4 Uart Core

Modulul de UART implementat realizează conversia din serial în paralel a operanzilor primiți folosind transmisia serială. Acest modul, UART_FPGA, funcționează sincron cu un ceas cu frecvența de 50 MHz, și prevede un semnal de reset, asincron. Semnalul de intrare RX, conține informația provenită din puntea USB-to-UART disponibilă pe placă. Semnalul RX corespunde unui modul de recepție a datelor, de la calculator către FPGA. Modulul UART_FPGA este prevăzut cu 3 semnale de ieșire. Aceste semnale sunt: rx_data, semnal cu o lungime de 8 biți conține informația primită prin transmisie serială, în acest caz fiind reprezentată prin caractere in cod ASCII primite de la tastatura calculatorului. Semnalul rx_ready este asertat în momentul în care s-a terminat decodarea semnalului de intrare RX, sugerând că valoarea din registrul rx_data este completă și corectă. Semnalul TX corespunde unui modul de transmisie, de la FPGA la calculator. Acest modul face parte din dezvoltările ulterioare, întrucât prezenta lucrare necesită doar recepția datelor nu și transmisia acestora. Semnalul TX este instanțiat cu 1 logic, acest lucru sugerând că nu se transmite niciun octet de date.

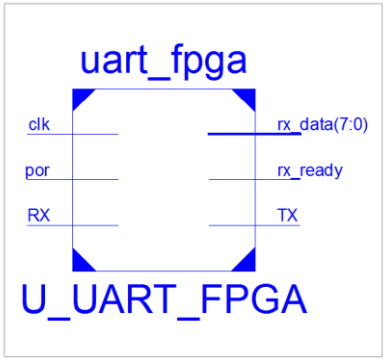

Fig. 4.9 Modul top – UART FPGA

4.1.2.5 Verificare, rezultate

Rezultatele simulării blocului ce asigura comunicația între calculator și cipul FPGA sunt ilustrate în figura 4.10. Fiind un modul sincron vom observa evenimentele semnalelor pe frontul crescător al semnalului de ceas, semnal cu frecvența 50 MHz. Semnalul RX este cel primit din puntea USB - to-UART, transmis serial. Acest semnal este preluat de modulul uart_fpga și decodat paralel. În figură se observă semnalul rx_data_int ce reprezintă registrul în care se va încarca rezultatul obținut dupa decodarea semnalului RX. De asemenea, se poate observa și semnalul rx_ready_int ce este asertat în momentul în care s-a încheiat transmisia și registrul rx_data_int a fost încarcat cu biții de date primiți pe UART.

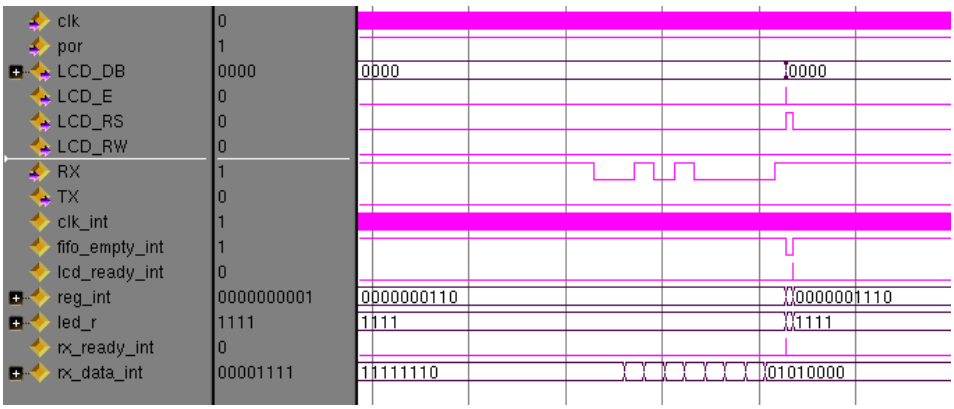

Fig. 4.10 Simulare modul UART

4.1.3 BLOC ÎNMULȚIRE CU ARITMETICĂ REDUNDANTĂ

Blocul rb_multiplicator este un modul principal ce are două semnale de intrare, rx_ready și rx_data, și două semnale de ieșire rb_ready și rb_data. Acest modul este sincron cu toate celelalte submodule conținute de modulul principal. De asemenea, este prevăzut și cu un semnal de reset, asincron. Funcționalitatea modulului rb_multiplicator este de a prelua datele recepționate de modulul uart_fpga, decodificarea acestora din cod ASCII în reprezentare hexazecimală și obținerea celor doi operanzi necesari pentru înmulțire. După ce s-a obținut rezultatul înmulțirii este urmat procedeul invers celui de la intrare, și anume: codificarea operanzilor din reprezentare hexazecimală în cod ASCII și trimiterea acestora către modulul de memorie, urmând să fie trimise datele către afișare.

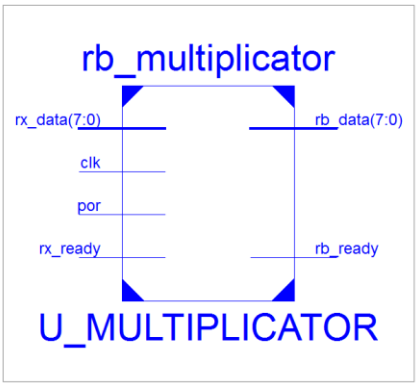

Fig. 4.11 Modul top - rb_multiplicator

Modulul rb_multiplicator conține 4 blocuri componente: un decodor din cod ASCII în reprezentare hexazecimală, decode_ascii, modulul de înmulțire cu aritmetică redundantă, mply_top, un decodor din reprezentare hexazecimală în cod ASCII, decode_hex și un numărător, counter.

Atât decodorul din cod ASCII în reprezentare hexazecimală, cât și decodorul din reprezentare hexazecimală în cod ASCII îndeplinesc funcția unei memorii ROM. Componenta decode_ascii este o memorie ROM cu adrese de lungime 8 biți și un numar de 2^8 locații de memorie de dimensiune de 4 biți. Componenta decode_ascii este o memorie ROM cu adrese de lungime 4 biți și un numar de 2^4 locații de memorie de dimensiune de 8 biți.

Componenta counter este un numărător care are funcția de a genera o întârziere din momentul decodării operanzilor până la începerea decodării rezultatului.

Componenta mply_top reprezintă blocul de înmulțire cu aritmetică redundantă despre care s-a discutat în subcapitolul 3.6.

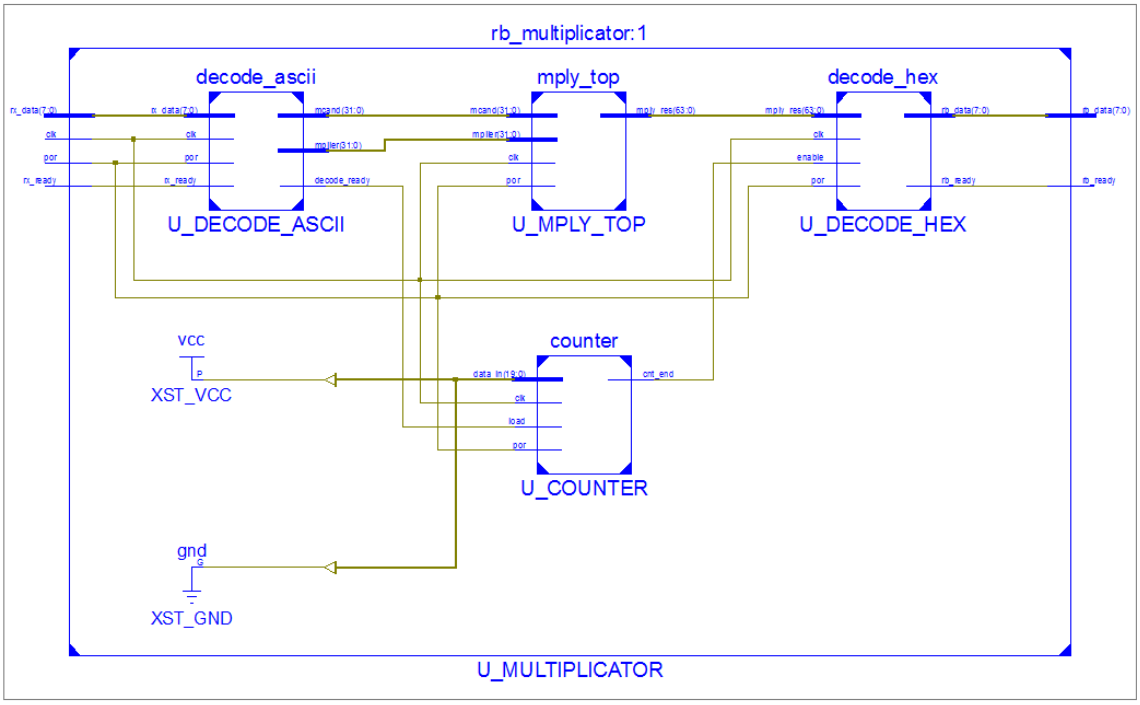

Fig. 4.12 Modul rb_multiplicator - blocuri componente

4.1.4 MEMORIE DE STOCARE A DATELOR

Datele ce urmează să fie trimise spre afișare sunt stocate într-un bloc de memorie. Acest bloc este organizat sub formă de registre, iar primul registru încărcat cu date va fi citit primul. Acest bloc de memorie este în esență o structură FIFO.

Modulul fifo este sincron cu celelalte submodule componente ale modulului principal top_multiplicator și este prevăzut și cu un semnal de reset, asincron. La intrare are cinci semnale din care două sunt semnale de date, rx_data, provenit de la submodulul uart_fpga și semnalul rb_data provenit de la submodulul rb_multiplicator. Aceste semnale au o lungime de 8 biți. Celelalte trei semnale sunt: rx_ready, care semnifică terminarea încărcării registrului cu datele provenite din transmisia serială.

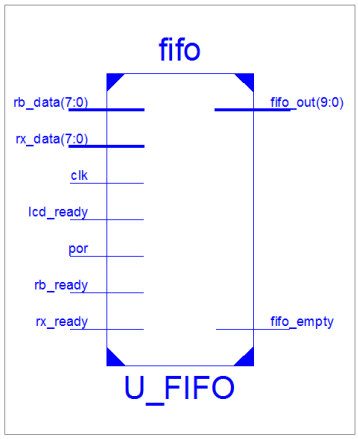

Fig. 4.13 Modul top – fifo

În figura 4.13 sunt evidențiate formele de undă ce demonstrează buna funcționare a modulului fifo. În registrul reg_int sunt încărcate valorile ce urmează să fie trimise spre modulul lcd_init în vederea afișării. În simulare a fost selectată afișarea valorilor registrului reg_int codat ASCII pentru a putea verifica facil ceea ce se așteaptă să fie trimis și ceea ce s-a trimis efectiv.

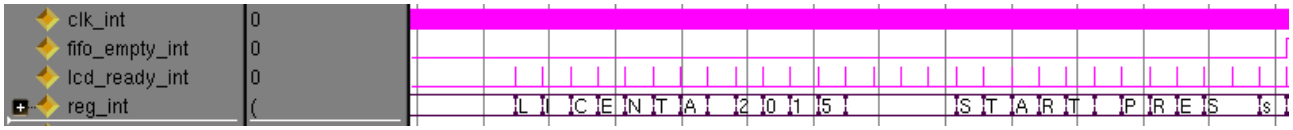

Fig. 4.14 Simulare - modul fifo

4.1.5 PROTOCOL DE COMUNICAȚIE CU AFIȘAJUL

Placa de dezvoltare VC707 folosește interfață de 4 biți pentru comunicarea cu afișajul LCD. În figura este ilustrată operația de scriere a afișajului, fiind evidențiate perioadele minime de timp necesare pentru setup, hold și durata de impuls, raportate la un semnal de ceas cu frecvența de 50 MHz (perioadă de 20 ns).

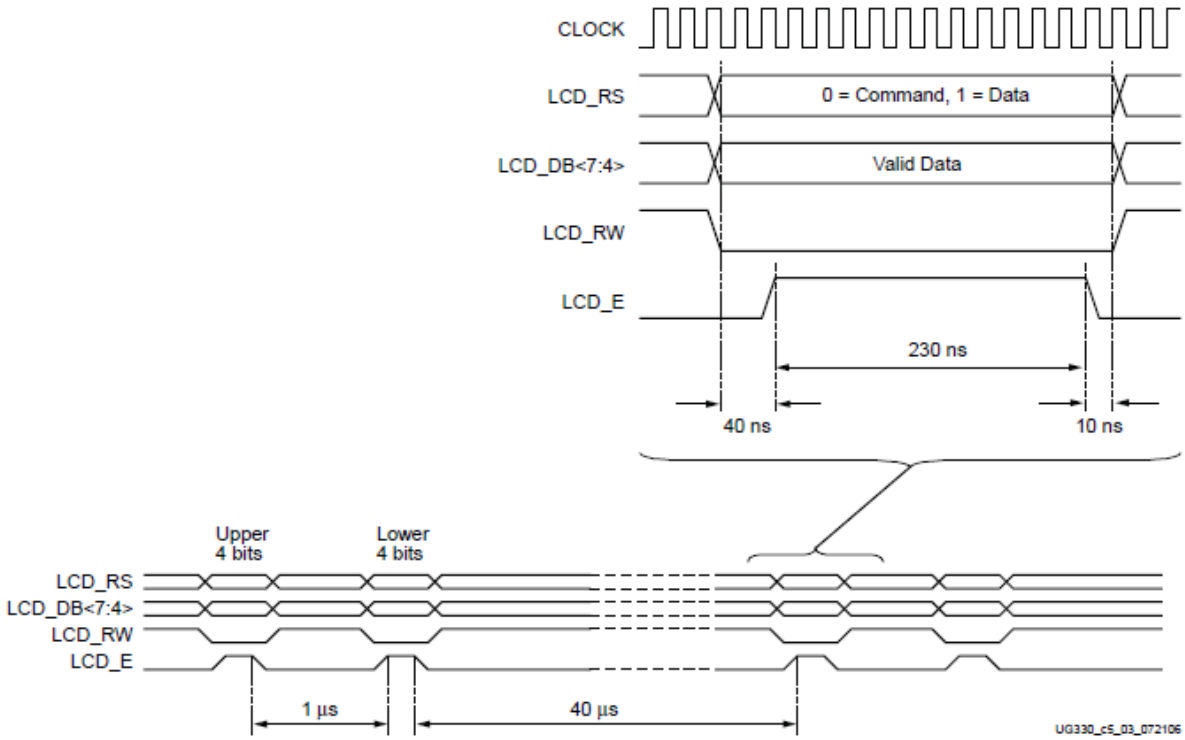

Fig. 4.15 Operația de scriere a afișajului [12]

Valorile semnalului de date (LCD_DB<7:4>), semnalului de selectare registru (LCD_RS) și semnalului de control citire/scriere (LCD_RW) trebuie sa fie stabile cel puțin o perioada de 40ns, înainte de asertarea semnalului de activare LCD_E. Acest lucru înseamnă respectarea timpului de setup. Semnalul de activare trebuie sa fie activ în 1 logic cel puțin o perioada de 230 ns (echivakentul a 12 perioade de ceas cu frecvența de 50 MHz). La sfârșitul perioadei minime de 230 ns, este necesar o perioada de timp de așteptare de cel puțin 10 ns, apoi semnalul de activare este trecut în 0 logic, iar semnalele LCD_RS, LCD_RW, LCD_DB<7:4> pot fi modificate.

În numeroase aplicații, semnalul de control citire/scriere este 0 (Low) permanent, întrucât, în general FPGA-ul nu are nevoie sa citească date de la afișaj.

În cazul în care semnalul de activare LCD_E este 0 (Low), toate semnalele de intrare ale afișajului sunt ignorate.

Transferul a 8 biți de date folosind o interfață de transfer de 4 biți este realizată prin trimiterea niblului superior de date, urmat de niblul inferior de date. Între transferul celor 2 nibli de date, este necesar un timp de așteptare de cel puțin 1 μs. Pentru ca urmatoarea operație de transfer de date să se realizeze corect, este necesar un timp de așteptare de 40 μs între sfârșitul transferului primului octet de date și începutul transferului celui de-al doilea octet de date. Perioada de asteptare de 40 μs se poate extinde până la 1.64 ms, când este urmată de o comanda de ștergere a afișajului (Clear Display).

În vederea configurării afișajului sunt necesari trei pași: inițializare la alimentare (Power-on initialisation), configurare și afișare.

Inițializarea la alimentare este necesară pentru stabilirea protocolului de comunicație. Secvența de inițializarea este sub forma unui automat de stari, iar pașii ce trebuie respectați sunt următorii:

1. Timp de așteptare de 15 ms sau mai mult; o perioadă de 15 ms înseamnă 750 000 de perioade de ceas cu frecvența de 50 MHz
2. Scrierea LCD_DB<7:4> = 0x3 și LCD_E = 1 timp de 12 perioade de ceas cu frecvența de 50 MHz
3. Timp de așteptare de 4.1 ms sau mai mult; o perioadă de 4.1 ms înseamnă 205 000 de perioade de ceas cu frecvența de 50 MHz
4. Scrierea LCD_DB<7:4> = 0x3 și LCD_E = 1 timp de 12 perioade de ceas cu frecvența de 50 MHz
5. Timp de așteptare de $100\ \mu s$ sau mai mult; o perioadă de $100\ \mu s$ înseamnă 5 000 de perioade de ceas cu frecvența de 50 MHz
6. Scrierea LCD_DB<7:4> = 0x3 și LCD_E = 1 timp de 12 perioade de ceas cu frecvența de 50 MHz
7. Timp de așteptare de $40\ \mu s$ sau mai mult; o perioadă de $100\ \mu s$ înseamnă 2 000 de perioade de ceas cu frecvența de 50 MHz
8. Scrierea LCD_DB<7:4> = 0x2 și LCD_E = 1 timp de 12 perioade de ceas cu frecvența de 50 MHz
9. Timp de așteptare de $40\ \mu s$ sau mai mult; o perioadă de $100\ \mu s$ înseamnă 2 000 de perioade de ceas cu frecvența de 50 MHz

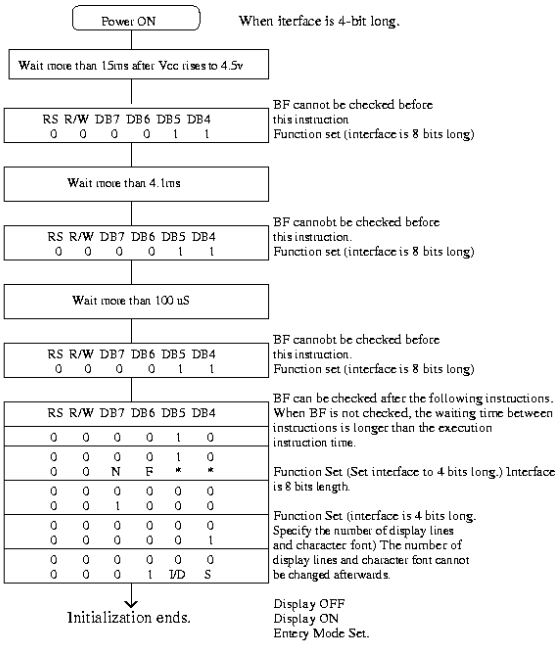

Fig. 4.16 Automatul de stări pentru inițializarea afișajului [16]

Dupa finalizarea etapei de inițializare la alimentare este stabilit protocolul de comunicație cu afișajul printr-o interfață de transfer de 4 biți. Urmează etapa de configurare ce presupune următoarea secvență de comenzi:

1. Apelarea unei comenzi de setare a funcțiilor (Function Set), 0x28, în vederea configurării afișajului corespunzător kit-ului de dezvoltare VC707
2. Apelarea unei comenzi de setare a modului de introducere (Entry Mode Set), 0x06, în vederea setării numărătorului de adrese al memoriei DD RAM să se auto-incrementeze.
3. Apelarea unei comenzi de aprindere/stingere afișaj (Display On/Off), 0x0C, pentru a aprinde afișajul și a dezactiva cursorul și clipirea acestuia.
4. Ultima comandă necesara este cea de ștergere a afișajului (Clear Display). După apelul acestei comenzi este necesar un perioadă de înârziere cde cel puțin 1.64 ms (82 000 de perioade de ceas cu frecvența de 50 MHz)

Odata ce etapa de configurare a afișajului este completă, se pot scrie date pe acesta. Pentru a putea face acest lucru este necesară specificarea adresei de început, urmată de mai multe valori de date. Înaintea scrierii datelor, este necesară apelarea unei comenzi de setare a adresei memoriei DD RAM (Set DD RAM Address) pentru a specifica adresa de 8 biți inițială. În subcapitolul 2.4.3 în figura 2.13 este ilustrată memoria DD RAM cu locațiile și adresele corespunzătoare.

Pentru scrierea datelor pe afișaj este necesară utilizarea unei comenzi de scriere în CG RAM sau DD RAM (Write Data to CG RAM or DD RAM). Cei 8 biți de date reprezintă valoarea adresei din CG ROM (valoarea în ASCII a caracterelor) sau controleaza matricea de puncte 5x8 pentru a reprezenta caracterul asociat.

Dacă numărătorul de adrese al memoriei DD RAM a fost configurat pentru a se auto-incrementa, așa cum a fost descris mai devreme, aplicația poate scrie succesiv mai multe caractere, iar fiecare caracter este scris automat și afișat în urmatoarea locație disponibilă.

Scrierea continuă a caracterelor, însă, presupune ajungerea la sfârșitul primei linii a afișajului. Caracterele ce vor fi scrise în continuare nu se vor scrie automat pe cea de-a doua linie deoarece adresele celor două linii nu sunt consecutive, apărând astfel necesitatea apelări unei comenzi de setare a adresei memoriei DD RAM (Set DD RAM Address) pentru a specifica adresa primei locații a celui de-al doilea rand.

În figura 4.16 este ilustrată schema bloc a submodulul lcd_init. Acest submodul este sincronizat cu un semnal de ceas cu frecvența de 50 MHz, semnal de ceas provenit de la modulul ierarhic superior, top_multiplicator. De asemenea, submodulul prevede și un semnal de reset asincron, comun pentru toate modulele componente modulului top_multiplicator. După cum se observă în schema bloc, avem două semnale de intrare, reg cu o lungime de 10 biți și fifo_empty. Semnalele de ieșire sunt lcd_ready, LCD_E, LCD_RS, LCD_RW și LCD_DB cu o lungime de 4 biți.

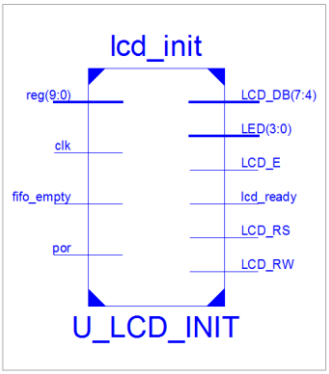

Fig. 4.17 Modul LCD pentru inițializare și afișare

Funcționalitatea submodulului lcd_init este de a realiza comunicația cu afișajul. Logica acestui modul constă într-un automat finit de stări, împărțit în două etape. Prima etapă constă în inițializarea afișajului la alimentare, aceasta fiind îndeplinită parcurgând o singură dată succesiunea de stări cu întârzierile corespunzătoare prezentată mai sus. Întârzierile sunt asigurate de un modul cu funcție de numărător, counter, ce are ca scop incrementarea unei variabile până se ajunge la o limită impusă, aferentă fiecărui pas din etapa de inițializare. Odată cu sfârșitul etapei de inițializare, un semnal intern, config_ready, este asertat semnificând astfel automatului că se poate trece la etapa următoare. Cea de-a doua etapă presupune respectarea rutinei de afișare pe LCD. Așa cum a fost explicat anterior, pentru a realiza corect operația de afișare este necesară respectarea formelor de undă prezentate în figura 4.14. Aceasta parte a automatului de stări este general valabilă și va fi folosită în continuare, ori de câte ori va exista necesitatea de a afișa pe LCD. Un lucru important ce trebuie precizat este că în momentul în care vor exista două cereri simultane de afișare nu se va pierde informație, întrucât semnalul lcd_ready este menit să semnaleze submodulului fifo când se pot trimite date spre afișare. Astfel, din modulul fifo se vor prelua date doar când semnalul lcd_ready este activ în 1 logic și submodulul lcd_init este disponibil pentru primirea acestora și prelucrarea acestora.

4.2 REZULTATELE SIMULĂRII

În acest capitol se vor explica rezultatele simulărilor blocurilor implementate pentru realizarea practică utilizând placa de dezvoltare VC707 cu cip FPGA Virtex-7.

Așa cum s-a discutat în capitolul 4, componentele acestui bloc principal ce realizează înmulțirea în reprezentare în aritmetică redundantă sunt:

- bloc de divizare a frecvenței de la 200MHz la 50 MHz
- bloc ce asigura transmisia serială și conversia datelor din serial în paralel
- bloc de stocare a datelor primite pe UART și a rezultatului înmulțirii
- blocul ce realizează înmulțirea operanzilor transmiși serial de utilizator de la tastatura calculatorului
- blocul ce asigura comunicația cu afișajul LCD

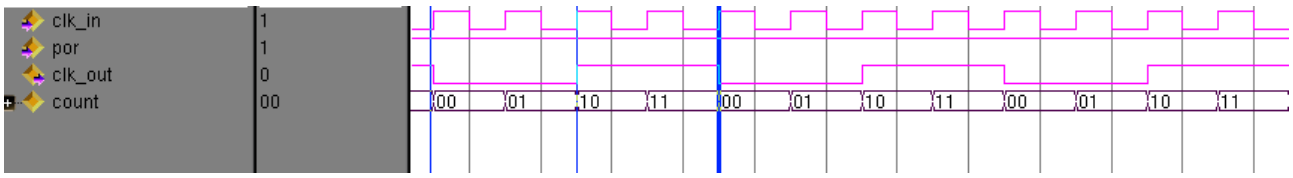

Fig. 4.18 Simulare - modul freq_div

Submodulul de divizare de frecvență freq_div are rolul de a micșora frecvența sistemului de 200 MHz de 4 ori, frecvența de 50 MHz obținută fiind necesară pentru a asigura buna funcționare a transmisiei seriale și a afișajului LCD. În figura 4.18 se poate observa această divizare de frecvență, unde se observă că semnalul de ceas obținut în urma divizării de frecvență are o perioadă de 4 ori mai mare decât cea a semnalului de ceas inițial.

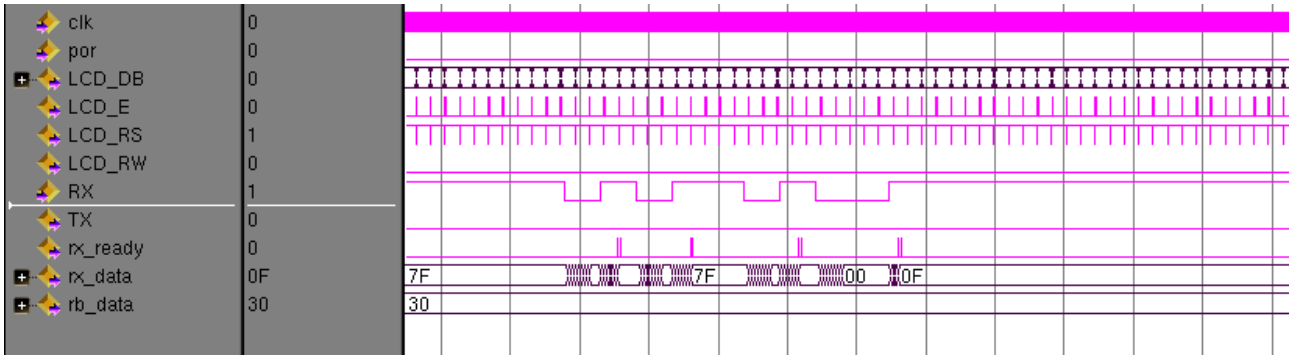

Fig. 4.19 Simulare - module UART și FIFO

Rezultatele simulării blocului ce asigura comunicația între calculator și cipul FPGA sunt ilustrate în figura 4.19. Semnalul RX este cel primit din puntea USB - to- UART, transmis serial. Acest semnal este preluat de modulul uart_fpga și decodat paralel. În figură se observă semnalul rx_data ce reprezintă registrul în care se va încarca rezultatul obținut după decodarea semnalului RX. De asemenea, se poate observa și semnalul rx_ready ce este asertat în momentul în care s-a încheiat transmisia și registrul rx_data a fost încarcat cu biții de date primiți pe UART.

Fig. 4.20 Simulare - modul rb_ multiplicator

Rezultatele simulării blocului de înmulțire cu aritmetică redundantă sunt evidențiate în figura 4.20. Ținând cont că modulul este sincron se vor urmări evenimentele semnalelor în momentul apariției frontului crescător al ceasului. Primul cursor marchează încărcarea operanzilor în registre pe frontul crescător al ceasului. Cel de-al doilea cursor evidențiază rezultatul înmulțirii obținut după o perioadă de ceas.

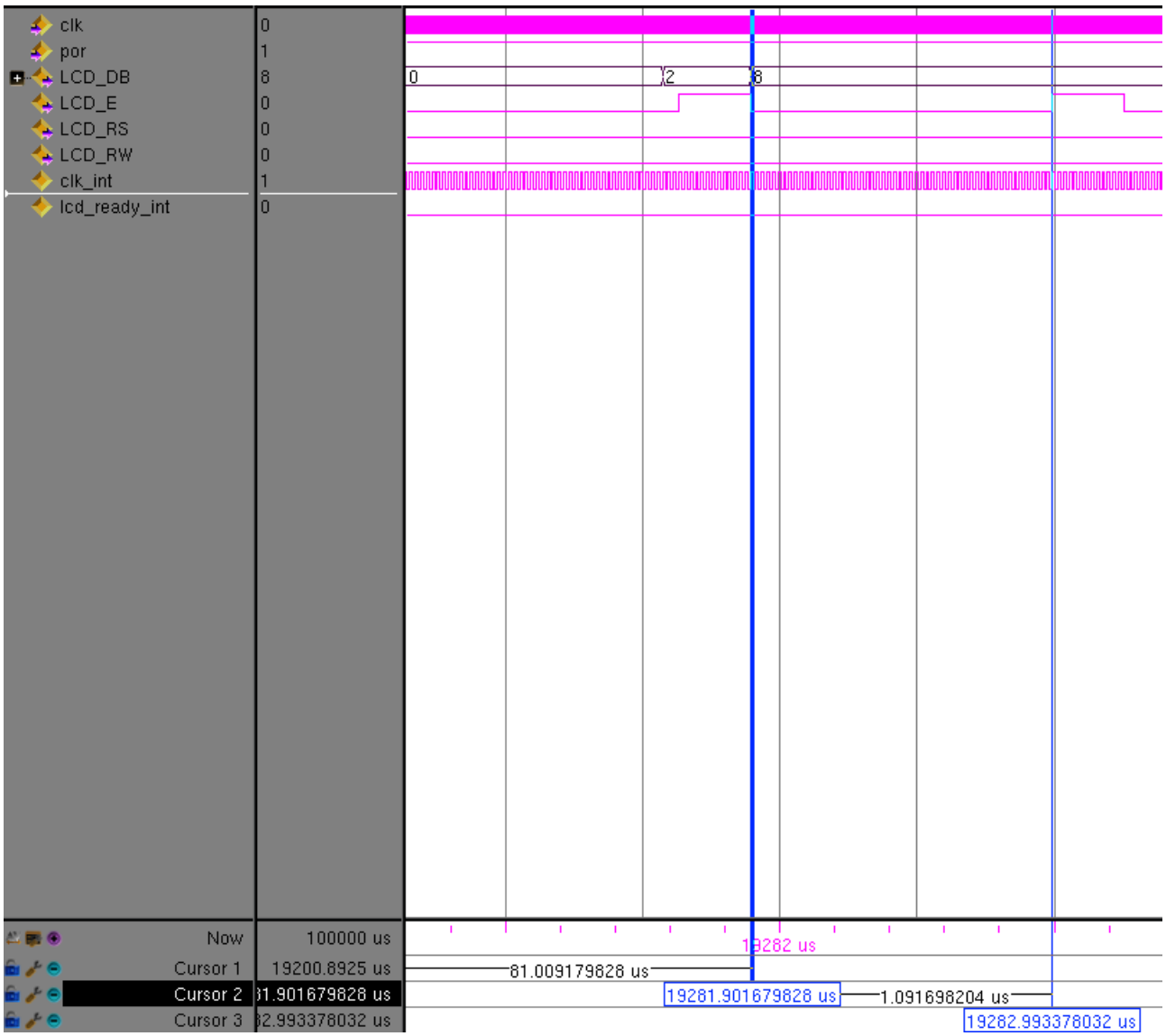

Fig. 4.21 Simulare - modul lcd

Așa cum am discutat în subcapitolul 4.1.5, în vederea stabilirii comunicației cu afișajul este necesară respectarea unor întârzieri. Astfel, în figura 4.21 este evidențiat timpul necesar de așteptare între două pachete de 4 biți consecutive de cel puțin 1 μs.

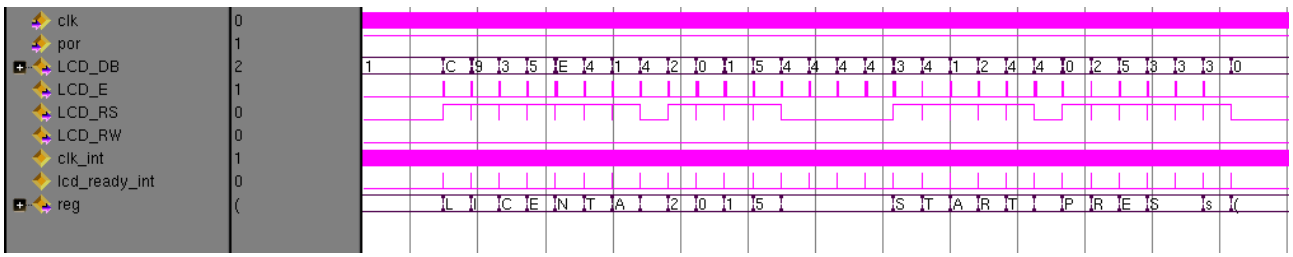

Fig. 4.22 Simulare - module LCD și FIFO

În figura 4.22 sunt evidențiate formele de undă ce demonstrează buna funcționare a modulelor fifo și lcd_init. În registrul reg sunt încărcate valorile ce urmează să fie trimise spre modulul lcd_init în vederea afișării. În simulare a fost selectată afișarea valorilor registrului reg codat ASCII pentru a putea verifica facil ceea ce se așteaptă să fie trimis și ceea ce s-a trimis efectiv. Semnalul LCD_DB arată trimiterea datelor către afișajul LCD în pachete de date de 4 biți, dat fiind faptul că se folosește o interfața de comunicație pe niblu.

CONCLUZII ȘI DEZVOLTĂRI ULTERIOARE

Utilizarea aritmeticii neconvenționale bazată pe aritmetica redundantă a devenit posibilă recent. Din aceste considerente, implementarea unor unități aritmetice de acest gen este încă în stadiul de dezvoltare.

Principalul obiectiv al acestei lucrări a fost implementarea unui bloc de înmulțire cu aritmetică redundantă capabil sa livreze rezultatul înmulțirii într-o perioadă de ceas de la primirea operanzilor.

S-a urmărit evidențierea pașilor necesari implementării acestui bloc de înmulțire, cu emfază pe cele mai importante noțiuni și blocuri componente, și anume *Blocul generator de produse parțiale* și *Blocul de însumare de tip Wallace Tree*, importante în obținerea rezultatului înmulțirii într-o singură perioadă de ceas.

În vederea îndeplinirii obiectivului propus am parcurs numeroși pași, prezentați în cadrul acestei lucrări. Contribuția personală a autorului se poate rezuma astfel:

- am realizat un studiu detaliat al modului de lucru folosind un tip de aritmetică neconvențională
- am găsit o soluție optimă astfel încât implementarea propriu-zisă a acestui tip de aritmetică să aducă îmbunătățiri față de aritmetica normală
- am proiectat un bloc cu aritmetică normală și un bloc cu aritmetică redundantă pentru a putea analiza performanțele acestora
- am dezvoltat blocurile necesare în vederea implementării practice a blocului de înmulțire ce utilizează aritmetică redundantă utilizând o placă de dezvoltare de la firma Xilinx, cu cip FPGA Virtex- 7

Rezultatele simulărilor evidențiază, în cazul ideal, modul de funcționare al blocului principal implementat pe placa de dezvoltare cu cip FPGA Virtex-7. În cazul real însă, am întâmpinat probleme la funcționare. În urma încercărilor de depanare cu ajutorul osciloscopului problema pare să vină de la modulul de memorie fifo. În viitor se dorește reproiectarea acestui modul și testarea funcționalității întregului sistem în condiții reale.

La dezvoltările ulterioare ale acestui proiect se încadrează următoarele:

- reproiectarea unor submodule în vederea folosirii blocului de înmulțire pentru operanzi de orice dimensiune
- implementarea unui bloc de înmulțire cu aritmetică redundantă cu alt tip de codare, de exemplu, codare pozitiv – negativ
- implementarea unui bloc de înmulțire cu aritmetică redundantă care lucrează în virgulă mobilă

BIBLIOGRAFIE

[1] Kharbash, F. Q., “Redundant adder architectures for cell-based technology”, Ph.D. Thesis, University of Missouri-Kansas City, 2011

[2] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi, „A Highspeed Multiplier Using a Redundant Binary Adder Tree”, IEEE Journal of Solid-State Circuits, vol. 22, no. 1, pp. 28-34, 1987.

[3] T. N. S. Kuninobu and T. Taniguchi, „High speed MOS Multiplier and Divider Using Redundant Binary Representation and Their Implementation in a Microprocessor”,

IEICE Transcations on Electronics, vol. E76-C, no. 3, pp. 436-445, 1993.

[4] H. Makino, Y. Nakase, and H. Shinohara, „ A 8.8-ns 54 x 54-bit Multiplier Using New Redundant Binary Architecture”, in the Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '93), 1993, pp. 202- 205.

[5] Y. Kim, B.-S. Song, J. Grosspietsch, and S. Gillig, „ A Carry-free 54 x 54b Multiplier Using Equivalent Bit conversion Algorithm”, IEEE Journal of Solid- State Circuits, vol. 36, no. 10, pp. 1538- 1545, 2001.

[6] Y. Takahashi, K. ichi Konta, K. Takahashi, M. Yokoyama, K. Shouno, and M. Mizunuma, „Carry Propagation Free Adder/Subtracter Using Adiabatic Dynamic CMOS Logic Circuit Technology”, IEICE Transcations on Fundamentals of Electronics, Communications and Computer Sciences, vol. E86-A, no. 6, pp. 1437-1444, 2003.

[7] K. Sarrigeorgidis and J. Rabaey, „Ultra Low Power CORDIC Processor for Wireless Communication Algorithms”, The Journal of VLSI Signal Processing, vol. 38, no. 2, pp. 115-130, 2004.

[8] E. Sava_s, „A Carry-free Architecture for Montgomery Inversion” IEEE Transactions on Computers, vol. 54, no. 12, pp. 1508-1519, 2005.

[9] http://wiki.dcae.pub.ro/index.php/Introducere_%C3%AEn_sinteza_pe_FPGA._Xilinx_ISE

[10] https://en.wikipedia.org/wiki/Xilinx_ISE

[11] VC707 Evaluation Board for the Virtex-7 FPGA User Guide, UG885, 2014
http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf

[12] Spartan -3A/3AN FPGA Starter Kit Board User Guide, UG334, 2008
http://www.gta.ufrj.br/ensino/EEL480/spartan3/ug334.pdf

[13] http://en.wikipedia.org/wiki/Serial_port#DTE_and_DCE

[14] http://users.utcluj.ro/~baruch/sie/labor/Port-Serial.pdf

[15] He, Y., & Chang, C. H. (2009). „ New Redundant Binary Booth Encoding for Fast 2^n-bit Multiplier Design". IEEE Transactions On Circuits And Systems—I. 56(6), 1192-1201

[16] 162D Series LCD Module Produc Specification , 2009, http://cdn.displaytech-us.com/sites/default/files/display-data-sheet/162D%20series-v31.pdf

[17] 7-Series FPGAs SelectIO Resources, UG471, 2014, http://www.xilinx.com/support/documentation/user_guides/ug471_7Series_SelectIO.pdf

[18] https://phaminhduc.wordpress.com/project/embedded-system-development-04-uart/

[19] D. Chandel, G. Kumawat, P. Lahoty, V.V. Chandrodaya, S. Sharma (2013); „Booth Multiplier: Ease of mmultiplication". IJETAE, ISO 9001:2008 Certified Journal, Volume 3, Issue 3

[20] http://www.geoffknagge.com/fyp/carrysave.shtml

ANEXA 1

Codul sursă pentru blocul de înmulțire ce utilizează aritmetica redundantă

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity mply_top is

port(
        por                     : in  std_ulogic;
        clk                     : in  std_ulogic;
        mplier: in  std_ulogic_vector(31 downto
0);
        mcand  : in  std_ulogic_vector(31 downto
0);
        mply_res: out std_ulogic_vector(63
downto 0)
        );
end mply_top;

architecture rtl of mply_top is

signal   pp1_1: std_ulogic_vector(34 downto 0);
signal   pp1_0: std_ulogic_vector(34 downto 0);
signal   pp2_1: std_ulogic_vector(34 downto 0);
signal   pp2_0: std_ulogic_vector(34 downto 0);
signal   pp3_1: std_ulogic_vector(34 downto 0);
signal   pp3_0: std_ulogic_vector(34 downto 0);
signal   pp4_1: std_ulogic_vector(34 downto 0);
signal   pp4_0: std_ulogic_vector(34 downto 0);
signal   pp5_1: std_ulogic_vector(34 downto 0);
signal   pp5_0: std_ulogic_vector(34 downto 0);
signal   pp6_1: std_ulogic_vector(34 downto 0);
signal   pp6_0: std_ulogic_vector(34 downto 0);
signal   pp7_1: std_ulogic_vector(34 downto 0);
signal   pp7_0: std_ulogic_vector(34 downto 0);
signal   pp8_1: std_ulogic_vector(34 downto 0);
signal   pp8_0: std_ulogic_vector(34 downto 0);
signal   rba_1: std_ulogic_vector(63 downto 0);
signal   rba_0: std_ulogic_vector(63 downto 0);
signal   mplier_reg: std_ulogic_vector(31 downto
0);
signal   mcand_reg: std_ulogic_vector(31 downto
0);
signal   mply_res_reg: std_ulogic_vector(63
downto 0);
signal   mply_res_out: std_ulogic_vector(63
downto 0);

component pp_gen
port(
        mplier:in std_ulogic_vector(31 downto 0);
        mcand:in std_ulogic_vector(31 downto
0);
        pp1_1:out std_ulogic_vector(34 downto
0);
        pp1_0:out std_ulogic_vector(34 downto
0);
        pp2_1:out std_ulogic_vector(34 downto
0);
        pp2_0:out std_ulogic_vector(34 downto
0);
        pp3_1:out std_ulogic_vector(34 downto
0);
        pp3_0:out std_ulogic_vector(34 downto
0);
        pp4_1:out std_ulogic_vector(34 downto
0);
        pp4_0:out std_ulogic_vector(34 downto
0);
        pp5_1:out std_ulogic_vector(34 downto
0);
        pp5_0:out std_ulogic_vector(34 downto
0);
        pp6_1:out std_ulogic_vector(34 downto
0);
        pp6_0:out std_ulogic_vector(34 downto
0);
        pp7_1:out std_ulogic_vector(34 downto
0);
        pp7_0:out std_ulogic_vector(34 downto
0);
        pp8_1:out std_ulogic_vector(34 downto
0);
        pp8_0:out std_ulogic_vector(34 downto
0)
        );
end component;

component wallace
port(
        pp1_1:in std_ulogic_vector(34 downto 0);
        pp1_0:in std_ulogic_vector(34 downto 0);
        pp2_1:in std_ulogic_vector(34 downto 0);
        pp2_0:in std_ulogic_vector(34 downto 0);
```

```
        pp3_1:in std_ulogic_vector(34 downto 0);
        pp3_0:in std_ulogic_vector(34 downto 0);
        pp4_1:in std_ulogic_vector(34 downto 0);
        pp4_0:in std_ulogic_vector(34 downto 0);
        pp5_1:in std_ulogic_vector(34 downto 0);
        pp5_0:in std_ulogic_vector(34 downto 0);
        pp6_1:in std_ulogic_vector(34 downto 0);
        pp6_0:in std_ulogic_vector(34 downto 0);
        pp7_1:in std_ulogic_vector(34 downto 0);
        pp7_0:in std_ulogic_vector(34 downto 0);
        pp8_1:in std_ulogic_vector(34 downto 0);
        pp8_0:in std_ulogic_vector(34 downto 0);

        rba_1: out std_ulogic_vector(63 downto
0);
        rba_0: out std_ulogic_vector(63 downto
0)
        );
end component;

begin


input_reg_proc : process(por,clk)
begin
 if por = '0' then
   mplier_reg    <= (others => '0');
   mcand_reg    <= (others => '0');
   mply_res_reg  <= (others => '0');
 elsif clk'event and clk = '1'  then
   mplier_reg    <= mplier;
   mcand_reg    <= mcand;
   mply_res_reg  <= mply_res_out;
 end if;
end process;

pp_gen_inst: pp_gen
 port map(
        mplier  => mplier_reg,
        mcand           => mcand_reg,
        pp1_1           => pp1_1,
        pp1_0           => pp1_0,
        pp2_1   => pp2_1,
        pp2_0   => pp2_0,
        pp3_1           => pp3_1,
        pp3_0           => pp3_0,
        pp4_1           => pp4_1,
        pp4_0           => pp4_0,
        pp5_1           => pp5_1,
        pp5_0           => pp5_0,
        pp6_1           => pp6_1,
        pp6_0           => pp6_0,
        pp7_1           => pp7_1,
        pp7_0           => pp7_0,
        pp8_1           => pp8_1,
        pp8_0           => pp8_0
                );

wallace_inst: wallace
 port map(
        pp1_1   => pp1_1,
        pp1_0   => pp1_0,
        pp2_1   => pp2_1,
        pp2_0   => pp2_0,
        pp3_1   => pp3_1,
        pp3_0   => pp3_0,
        pp4_1   => pp4_1,
        pp4_0   => pp4_0,
        pp5_1   => pp5_1,
        pp5_0   => pp5_0,
        pp6_1   => pp6_1,
        pp6_0   => pp6_0,
        pp7_1   => pp7_1,
        pp7_0   => pp7_0,
        pp8_1   => pp8_1,
        pp8_0   => pp8_0,

        rba_1   => rba_1,
        rba_0           => rba_0
                );

--      mply_res <=     to_std_ulogic_vector(
unsigned (rba_1) - unsigned (rba_0));
--      mply_res <=
        to_stdulogicvector(to_stdlogicvector (
unsigned (rba_1) -  (unsigned (rba_0)));
        mply_res_out <=
        to_stdulogicvector( unsigned (rba_1) -
(unsigned (rba_0)));
        mply_res        <=      mply_res_reg;

end rtl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.math_real.all;
```

```
entity pp_gen is

port(
        mplier: in  std_ulogic_vector(31 downto
0);
        mcand: in  std_ulogic_vector(31 downto
0);
        pp1_1: out std_ulogic_vector(34 downto
0);
        pp1_0 : out std_ulogic_vector(34 downto
0);
        pp2_1 : out std_ulogic_vector(34 downto
0);
        pp2_0 : out std_ulogic_vector(34 downto
0);
        pp3_1 : out std_ulogic_vector(34 downto
0);
        pp3_0: out std_ulogic_vector(34 downto
0);
        pp4_1: out std_ulogic_vector(34 downto
0);
        pp4_0: out std_ulogic_vector(34 downto
0);
        pp5_1: out std_ulogic_vector(34 downto
0);
        pp5_0: out std_ulogic_vector(34 downto
0);
        pp6_1: out std_ulogic_vector(34 downto
0);
        pp6_0: out std_ulogic_vector(34 downto
0);
        pp7_1: out std_ulogic_vector(34 downto
0);
        pp7_0: out std_ulogic_vector(34 downto
0);
        pp8_1: out std_ulogic_vector(34 downto
0);
        pp8_0: out std_ulogic_vector(34 downto
0)
        );
end pp_gen;

architecture rtl of pp_gen is

component x5
port(
        mcand: in  std_ulogic_vector(31 downto
0);
        mcand_not       : in  std_ulogic_vector(31
downto 0);
        x5_1 : out std_ulogic_vector(34 downto
0);
        x5_0 : out std_ulogic_vector(34 downto
0)
        );
end component;

component one_pair_pp
port(
        mcand: in  std_ulogic_vector(31 downto
0);
        Mp      : in  std_ulogic_vector(4 downto
0);
        x5_1 : in  std_ulogic_vector(34 downto
0);
        x5_0 : in  std_ulogic_vector(34 downto
0);
        pp1 : out std_ulogic_vector(34 downto 0);
        pp0 : out std_ulogic_vector(34 downto 0)
        );
end component;

        signal  mcand_not: std_ulogic_vector(31
downto 0);
        signal  x5_1_int: std_ulogic_vector(34
downto 0);
        signal  x5_0_int: std_ulogic_vector(34
downto 0);
        signal   pp1_1_int: std_ulogic_vector(34
downto 0);
        signal   pp1_0_int: std_ulogic_vector(34
downto 0);
        signal   pp2_1_int: std_ulogic_vector(34
downto 0);
        signal   pp2_0_int: std_ulogic_vector(34
downto 0);
        signal   pp3_1_int: std_ulogic_vector(34
downto 0);
        signal   pp3_0_int: std_ulogic_vector(34
downto 0);
        signal   pp4_1_int: std_ulogic_vector(34
downto 0);
        signal   pp4_0_int: std_ulogic_vector(34
downto 0);
        signal   pp5_1_int: std_ulogic_vector(34
downto 0);
```

```
        signal   pp5_0_int: std_ulogic_vector(34
downto 0);
        signal   pp6_1_int: std_ulogic_vector(34
downto 0);
        signal   pp6_0_int: std_ulogic_vector(34
downto 0);
        signal   pp7_1_int: std_ulogic_vector(34
downto 0);
        signal   pp7_0_int: std_ulogic_vector(34
downto 0);
        signal   pp8_1_int: std_ulogic_vector(34
downto 0);
        signal   pp8_0_int: std_ulogic_vector(34
downto 0);

        signal   Mp1: std_ulogic_vector(4 downto
0);
        signal   Mp2: std_ulogic_vector(4 downto
0);
        signal   Mp3: std_ulogic_vector(4 downto
0);
        signal   Mp4: std_ulogic_vector(4 downto
0);
        signal   Mp5: std_ulogic_vector(4 downto
0);
        signal   Mp6: std_ulogic_vector(4 downto
0);
        signal   Mp7: std_ulogic_vector(4 downto
0);
        signal   Mp8: std_ulogic_vector(4 downto
0);

begin

        mcand_not_proc: process (mcand)
        begin
        for i in 0 to 31 loop
        mcand_not(i) <= not(mcand(i));
        end loop;
        end process;

x5_inst: x5
        port map(
        mcand                   => mcand,
        mcand_not       => mcand_not,
        x5_1                    => x5_1_int,
        x5_0                    => x5_0_int
                );

        Mp1     <= mplier(3 downto 0) & '0';

one_pair_pp_1_inst: one_pair_pp
 port map(
        mcand                   => mcand,
        Mp                              => Mp1,
        x5_1                    => x5_1_int,
        x5_0                    => x5_0_int,
        pp1                     => pp1_1_int,
        pp0                     => pp1_0_int
                );

        pp1_1 <= pp1_1_int;
        pp1_0 <= pp1_0_int;


        Mp2     <= mplier(7 downto 3);

one_pair_pp_2_inst: one_pair_pp
 port map(
        mcand                   => mcand,
        Mp                              => Mp2,
        x5_1                    => x5_1_int,
        x5_0                    => x5_0_int,
        pp1                     => pp2_1_int,
        pp0                     => pp2_0_int
                );

        pp2_1 <= pp2_1_int;
        pp2_0 <= pp2_0_int;


        Mp3     <= mplier(11 downto 7);

one_pair_pp_3_inst: one_pair_pp
 port map(
        mcand                   => mcand,
        Mp                              =>
        Mp3,
        x5_1                    => x5_1_int,
        x5_0                    => x5_0_int,
        pp1                     => pp3_1_int,
        pp0                     => pp3_0_int
                );

        pp3_1 <= pp3_1_int;
        pp3_0 <= pp3_0_int;
```

```
	Mp4	<= mplier(15 downto 11);

one_pair_pp_4_inst: one_pair_pp
 port map(
	mcand			=> mcand,
	Mp				=>
	Mp4,
	x5_1			=> x5_1_int,
	x5_0			=> x5_0_int,
	pp1			=> pp4_1_int,
	pp0			=> pp4_0_int
		);

	pp4_1 <= pp4_1_int;
	pp4_0 <= pp4_0_int;


	Mp5	<= mplier(19 downto 15);

one_pair_pp_5_inst: one_pair_pp
 port map(
	mcand			=> mcand,
	Mp			=>	Mp5,
	x5_1			=> x5_1_int,
	x5_0			=> x5_0_int,
	pp1			=> pp5_1_int,
	pp0			=> pp5_0_int
		);

	pp5_1 <= pp5_1_int;
	pp5_0 <= pp5_0_int;


	Mp6	<= mplier(23 downto 19);

one_pair_pp_6_inst: one_pair_pp
 port map(
	mcand			=> mcand,
	Mp			=>	Mp6,
	x5_1			=> x5_1_int,
	x5_0			=> x5_0_int,
	pp1			=> pp6_1_int,
	pp0			=> pp6_0_int
		);

	pp6_1 <= pp6_1_int;
	pp6_0 <= pp6_0_int;

	Mp7	<= mplier(27 downto 23);

one_pair_pp_7_inst: one_pair_pp
 port map(
	mcand			=> mcand,
	Mp			=>	Mp7,
	x5_1			=> x5_1_int,
	x5_0			=> x5_0_int,
	pp1			=> pp7_1_int,
	pp0			=> pp7_0_int
		);

	pp7_1 <= pp7_1_int;
	pp7_0 <= pp7_0_int;


	Mp8	<= mplier(31 downto 27);

one_pair_pp_8_inst: one_pair_pp
 port map(
	mcand			=> mcand,
	Mp			=>	Mp8,
	x5_1			=> x5_1_int,
	x5_0			=> x5_0_int,
	pp1			=> pp8_1_int,
	pp0			=> pp8_0_int
		);

	pp8_1 <= pp8_1_int;
	pp8_0 <= pp8_0_int;

end rtl;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.math_real.all;

entity x5 is

port(
	mcand			: in
std_ulogic_vector(31 downto 0);
	mcand_not	: in  std_ulogic_vector(31
downto 0);
	x5_1			: out
std_ulogic_vector(34 downto 0);
```

```
        x5_0                    : out
std_ulogic_vector(34 downto 0)
        );
end x5;


architecture rtl of x5 is

        signal x4: std_ulogic_vector(34 downto
0);
        signal x1: std_ulogic_vector(34 downto
0);
        signal x5_0_1: std_ulogic;
        signal x5_0_0: std_ulogic;
        signal x5_1_1: std_ulogic;
        signal x5_1_0: std_ulogic;
        signal x5_2_1: std_ulogic;
        signal x5_2_0: std_ulogic;
        signal x5_1_int: std_ulogic_vector(34
downto 3);
        signal x5_0_int: std_ulogic_vector(34
downto 3);

begin


        x4      <= mcand(31) & mcand(31
downto 0) & "00";
                        --x1
        x1      <= mcand_not(31) &
mcand_not(31) & mcand_not(31) &
mcand_not(31 downto 0);         --x0


        x5_0_1  <= x1(0);
        x5_0_0 <= '1';

        x5_1_1  <= x1(1);
        x5_1_0 <= x1(0);

        x5_2_1  <= x4(2) xor x1(2);
        x5_2_0 <= x1(1);



        x5_proc: process (x4, x1)
        begin
        for i in 3 to 34 loop
        x5_1_int(i) <= x4(i) xor x1(i);
        x5_0_int(i) <= not x4(i-1) and x1(i-1);
        end loop;
        end process;

        x5_1    <= x5_1_int & x5_2_1 & x5_1_1
& x5_0_1;
        x5_0    <= x5_0_int & x5_2_0 & x5_1_0
& x5_0_0;

end rtl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.math_real.all;

entity one_pair_pp is

port(
        mcand  : in  std_ulogic_vector(31 downto
0);
        Mp      : in  std_ulogic_vector(4 downto
0);
        x5_1    : in  std_ulogic_vector(34 downto
0);
        x5_0    : in  std_ulogic_vector(34 downto
0);
        pp1     : out std_ulogic_vector(34 downto
0);
        pp0     : out std_ulogic_vector(34 downto
0)
        );
end one_pair_pp;

architecture rtl of one_pair_pp is

component Booth
port(
        Mp: in  std_ulogic_vector(4 downto 0);
        x1: out std_ulogic;
        x2: out std_ulogic;
        x4: out std_ulogic;
        x5: out std_ulogic;
        x8: out std_ulogic;
        inv : out std_ulogic
        );
end component;

        signal x1_int: std_ulogic;
        signal x2_int: std_ulogic;
```

```
        signal x4_int: std_ulogic;
        signal x5_int: std_ulogic;
        signal x8_int: std_ulogic;
        signal inv_int    : std_ulogic;

        signal m_8        : std_ulogic_vector(34
downto 0);
--      signal m_51       : std_ulogic_vector(34
downto 0);
--      signal m_50       : std_ulogic_vector(34
downto 0);
        signal m_4        : std_ulogic_vector(34
downto 0);
        signal m_2        : std_ulogic_vector(34
downto 0);
        signal m_1        : std_ulogic_vector(34
downto 0);
        signal m1         : std_ulogic_vector(34
downto 0);
        signal m0         : std_ulogic_vector(34
downto 0);
        signal m1_int     : std_ulogic_vector(34
downto 0);
        signal m0_int     : std_ulogic_vector(34
downto 0);

begin


Booth_inst: Booth
 port map(
        Mp              => Mp,
        x1              =>      x1_int,
        x2              =>      x2_int,
        x4              =>      x4_int,
        x5              =>      x5_int,
        x8              =>      x8_int,
        inv     =>      inv_int
                );

--      m_8 <= not mcand(31) & mcand(30
downto 0) & "000";
--      m_4 <= not mcand(31) & mcand(31
downto 0) & "00";
--      m_2 <= not mcand(31) & mcand(31) &
mcand & '0';
--      m_1 <= not mcand(31) & mcand(31) &
mcand(31) & mcand;
        m_8 <= mcand(31 downto 0) & "000";
        m_4 <= mcand(31) & mcand(31 downto
0) & "00";
        m_2 <= mcand(31) & mcand(31) &
mcand(31 downto 0) & '0';
        m_1 <= mcand(31) & mcand(31) &
mcand(31) & mcand(31 downto 0);



        m1_proc: process (m_8, m_4, x5_1,
x8_int, x4_int, x5_int)
        begin
        for i in 0 to 34 loop
        m1(i) <= (m_8(i) and x8_int) or (m_4(i)
and x4_int) or (x5_1(i) and x5_int);
        end loop;
        end process;

        m0_proc: process (m_2, m_1, x5_0,
x2_int, x1_int, x5_int)
        begin
        for i in 0 to 34 loop
        m0(i) <= (m_2(i) and x2_int) or (m_1(i)
and x1_int) or (x5_0(i) and x5_int);
        end loop;
        end process;

        m1_int <= not m1(34) & m1(33 downto
0);
        m0_int <= not m0(34) & m0(33 downto
0);

        pp1_proc: process (m1_int, m0_int,
inv_int)
        begin
        for i in 0 to 34 loop
        pp1(i) <= (m1_int(i) and (not inv_int)) or
(m0_int(i) and inv_int);
        end loop;
        end process pp1_proc;

        pp0_proc: process (m1_int, m0_int,
inv_int)
        begin
        for i in 0 to 34 loop
        pp0(i) <= (m0_int(i) and (not inv_int)) or
(m1_int(i) and inv_int);
        end loop;
        end process;
```

```
end rtl;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.math_real.all;

entity booth is

port(
        Mp : in  std_ulogic_vector(4 downto 0);
        x1 : out std_ulogic;
        x2 : out std_ulogic;
        x4 : out std_ulogic;
        x5 : out std_ulogic;
        x8 : out std_ulogic;
        inv      : out std_ulogic
        );
end booth;

architecture rtl of booth is

        signal Mp0      : std_ulogic;
        signal Mp1      : std_ulogic;
        signal Mp2      : std_ulogic;
        signal Mp3      : std_ulogic;
        signal Mp4      : std_ulogic;

begin
        Mp0     <= Mp(0);
        Mp1     <= Mp(1);
        Mp2     <= Mp(2);
        Mp3     <= Mp(3);
        Mp4     <= Mp(4);

--      x1      <= (Mp0 xor Mp1) and ((Mp4
xnor Mp3) or (Mp4 xor Mp2));
        x1      <= (Mp0 xor Mp1) and (not(Mp4
xor Mp3) or (Mp4 xor Mp2));
--      x2      <= (Mp(0) xnor Mp(1)) and
(Mp(1) xor Mp(2));
        x2      <= (not Mp0 and not Mp1 and
Mp2) or (Mp0 and Mp1 and not Mp2);
        x4      <= (Mp2 xor Mp3) and ((Mp4 xor
Mp2) or (not MP0 and not Mp1 and Mp3) or
(MP0 and Mp1 and not Mp3));
        x8      <= (Mp4 xor Mp3) and ((Mp4 xor
Mp2) or (MP0 and Mp1 and Mp3) or (not MP0
and not Mp1 and not Mp3));
        x5      <= (Mp4 xor Mp3) and (Mp3 xor
Mp2) and (MP1 xor Mp0);
        inv     <= (not Mp3 and not Mp2) or
(MP4 and (not Mp3 or not Mp2));

end rtl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
--use IEEE.math_real.all;

entity wallace is

port(
        pp1_1:  in std_ulogic_vector(34 downto
0);
        pp1_0:in std_ulogic_vector(34 downto 0);
        pp2_1:in std_ulogic_vector(34 downto 0);
        pp2_0:in std_ulogic_vector(34 downto 0);
        pp3_1:in std_ulogic_vector(34 downto 0);
        pp3_0:  in std_ulogic_vector(34 downto
0);
        pp4_1:  in std_ulogic_vector(34 downto
0);
        pp4_0:  in std_ulogic_vector(34 downto
0);
        pp5_1:  in std_ulogic_vector(34 downto
0);
        pp5_0:  in std_ulogic_vector(34 downto
0);
        pp6_1:  in std_ulogic_vector(34 downto
0);
        pp6_0:  in std_ulogic_vector(34 downto
0);
        pp7_1:  in std_ulogic_vector(34 downto
0);
        pp7_0:  in std_ulogic_vector(34 downto
0);
        pp8_1:  in std_ulogic_vector(34 downto
0);
        pp8_0:  in std_ulogic_vector(34 downto
0);

        rba_1: out std_ulogic_vector(63 downto
0);
```

```
        rba_0: out std_ulogic_vector(63 downto
0)
        );
end wallace;

architecture rtl of wallace is


component RBA_cell_PN
port(
        x1      : in  std_ulogic;
        x0      : in  std_ulogic;
        y1      : in  std_ulogic;
        y0      : in  std_ulogic;
        cin1    : in  std_ulogic;
        cin0    : in  std_ulogic;
        z1      : out std_ulogic;
        z0      : out std_ulogic;
        cy1     : out std_ulogic;
        cy0     : out std_ulogic
        );
end component;

        signal   zero                   :
std_ulogic;

        signal op01_1:std_ulogic_vector(39
downto 0);
        signal op01_0:std_ulogic_vector(39
downto 0);
        signal op02_1:std_ulogic_vector(39
downto 0);
        signal op02_0:std_ulogic_vector(39
downto 0);
        signal op03_1:std_ulogic_vector(39
downto 0);
        signal op03_0:std_ulogic_vector(39
downto 0);
        signal op04_1:std_ulogic_vector(39
downto 0);
        signal op04_0:std_ulogic_vector(39
downto 0);
        signal op05_1:std_ulogic_vector(39
downto 0);
        signal op05_0:std_ulogic_vector(39
downto 0);
        signal op06_1:std_ulogic_vector(39
downto 0);
        signal op06_0:std_ulogic_vector(39
downto 0);
        signal op07_1:std_ulogic_vector(39
downto 0);
        signal op07_0:std_ulogic_vector(39
downto 0);
        signal op08_1:std_ulogic_vector(39
downto 0);
        signal op08_0:std_ulogic_vector(39
downto 0);



        signal cy01_1:std_ulogic_vector(40
downto 1);
        signal cy01_0:std_ulogic_vector(40
downto 1);
        signal cy02_1:std_ulogic_vector(40
downto 1);
        signal cy02_0:std_ulogic_vector(40
downto 1);
        signal cy03_1:std_ulogic_vector(40
downto 1);
        signal cy03_0:std_ulogic_vector(40
downto 1);
        signal cy04_1:std_ulogic_vector(40
downto 1);
        signal cy04_0:std_ulogic_vector(40
downto 1);

        signal p11_1:std_ulogic_vector(39
downto 0);
        signal p11_0:std_ulogic_vector(39
downto 0);
        signal p12_1:std_ulogic_vector(39
downto 0);
        signal p12_0:std_ulogic_vector(39
downto 0);
        signal p13_1:std_ulogic_vector(39
downto 0);
        signal p13_0:std_ulogic_vector(39
downto 0);
        signal p14_1:std_ulogic_vector(39
downto 0);
        signal p14_0:std_ulogic_vector(39
downto 0);

        signal op11_1:std_ulogic_vector(47
downto 0);
```

```
        signal op11_0:std_ulogic_vector(47
downto 0);
        signal op12_1:std_ulogic_vector(47
downto 0);
        signal op12_0:std_ulogic_vector(47
downto 0);
        signal op13_1:std_ulogic_vector(47
downto 0);
        signal op13_0:std_ulogic_vector(47
downto 0);
        signal op14_1:std_ulogic_vector(47
downto 0);
        signal op14_0:std_ulogic_vector(47
downto 0);

        signal cy11_1:std_ulogic_vector(48
downto 1);
        signal cy11_0:std_ulogic_vector(48
downto 1);
        signal cy12_1:std_ulogic_vector(48
downto 1);
        signal cy12_0:std_ulogic_vector(48
downto 1);



        signal p21_1:std_ulogic_vector(47
downto 0);
        signal p21_0:std_ulogic_vector(47
downto 0);
        signal p22_1:std_ulogic_vector(47
downto 0);
        signal p22_0:std_ulogic_vector(47
downto 0);

        signal op21_1:std_ulogic_vector(63
downto 0);
        signal op21_0:std_ulogic_vector(63
downto 0);
        signal op22_1:std_ulogic_vector(63
downto 0);
        signal op22_0:std_ulogic_vector(63
downto 0);

        signal cy21_1:std_ulogic_vector(64
downto 1);
        signal cy21_0:std_ulogic_vector(64
downto 1);
        signal p31_1:std_ulogic_vector(63
downto 0);
        signal p31_0:std_ulogic_vector(63
downto 0);

begin

        zero    <= '0';


                -- level 0
        op01_1 <= "00000" & pp1_1;
        op01_0 <= "00000" & pp1_0;
        op02_1 <= '0' & pp2_1 & "0000";
        op02_0 <= '0' & pp2_0 & "0000";

lv01_0_inst     : RBA_cell_PN
 port map(
        x1      => op01_1(0),
        x0      => op01_0(0),
        y1      => op02_1(0),
        y0      => op02_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p11_1(0),
        z0      => p11_0(0),
        cy1     => cy01_1(1),
        cy0     => cy01_0(1)
                );

leve0_1: for i in 1 to 39 generate
lv01_i_inst     : RBA_cell_PN
 port map(
        x1      => op01_1(i),
        x0      => op01_0(i),
        y1      => op02_1(i),
        y0      => op02_0(i),
        cin1    => cy01_1(i),
        cin0    => cy01_0(i),
        z1      => p11_1(i),
        z0      => p11_0(i),
        cy1     => cy01_1(i+1),
        cy0     => cy01_0(i+1)
                );
end generate leve0_1;


        op03_1 <= "00000" & pp3_1;
        op03_0 <= "00000" & pp3_0;
```

```
        op04_1 <= '0' & pp4_1 & "0000";
        op04_0 <= '0' & pp4_0 & "0000";

lv02_0_inst     : RBA_cell_PN
 port map(
        x1      => op03_1(0),
        x0      => op03_0(0),
        y1      => op04_1(0),
        y0      => op04_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p12_1(0),
        z0      => p12_0(0),
        cy1     => cy02_1(1),
        cy0     => cy02_0(1)
                );

leve0_2: for i in 1 to 39 generate
lv02_i_inst     : RBA_cell_PN
 port map(
        x1      => op03_1(i),
        x0      => op03_0(i),
        y1      => op04_1(i),
        y0      => op04_0(i),
        cin1    => cy02_1(i),
        cin0    => cy02_0(i),
        z1      => p12_1(i),
        z0      => p12_0(i),
        cy1     => cy02_1(i+1),
        cy0     => cy02_0(i+1)
                );
end generate leve0_2;



        op05_1 <= "00000" & pp5_1;
        op05_0 <= "00000" & pp5_0;
        op06_1 <= '0' & pp6_1 & "0000";
        op06_0 <= '0' & pp6_0 & "0000";

lv03_0_inst     : RBA_cell_PN
 port map(
        x1      => op05_1(0),
        x0      => op05_0(0),
        y1      => op06_1(0),
        y0      => op06_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p13_1(0),
        z0      => p13_0(0),
        cy1     => cy03_1(1),
        cy0     => cy03_0(1)
                );

leve0_3: for i in 1 to 39 generate
lv03_i_inst     : RBA_cell_PN
 port map(
        x1      => op05_1(i),
        x0      => op05_0(i),
        y1      => op06_1(i),
        y0      => op06_0(i),
        cin1    => cy03_1(i),
        cin0    => cy03_0(i),
        z1      => p13_1(i),
        z0      => p13_0(i),
        cy1     => cy03_1(i+1),
        cy0     => cy03_0(i+1)
                );
end generate leve0_3 ;



        op07_1 <= "00000" & pp7_1;
        op07_0 <= "00000" & pp7_0;
        op08_1 <= '0' & pp8_1 & "0000";
        op08_0 <= '0' & pp8_0 & "0000";

lv04_0_inst     : RBA_cell_PN
 port map(
        x1      => op07_1(0),
        x0      => op07_0(0),
        y1      => op08_1(0),
        y0      => op08_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p14_1(0),
        z0      => p14_0(0),
        cy1     => cy04_1(1),
        cy0     => cy04_0(1)
                );

leve0_4: for i in 1 to 39 generate
lv04_i_inst     : RBA_cell_PN
 port map(
        x1      => op07_1(i),
        x0      => op07_0(i),
        y1      => op08_1(i),
        y0      => op08_0(i),
        cin1    => cy04_1(i),
```

```
        cin0    => cy04_0(i),
        z1      => p14_1(i),
        z0      => p14_0(i),
        cy1     => cy04_1(i+1),
        cy0     => cy04_0(i+1)
                );
end generate leve0_4 ;


                -- level 1
        op11_1 <= "00000000" & p11_1;
        op11_0 <= "00000000" & p11_0;
        op12_1 <= p12_1 & "00000000";
        op12_0 <= p12_0 & "00000000";

lv11_0_inst     : RBA_cell_PN
 port map(
        x1      => op11_1(0),
        x0      => op11_0(0),
        y1      => op12_1(0),
        y0      => op12_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p21_1(0),
        z0      => p21_0(0),
        cy1     => cy11_1(1),
        cy0     => cy11_0(1)
                );

leve1_1: for i in 1 to 47 generate
lv11_i_inst     : RBA_cell_PN
 port map(
        x1      => op11_1(i),
        x0      => op11_0(i),
        y1      => op12_1(i),
        y0      => op12_0(i),
        cin1    => cy11_1(i),
        cin0    => cy11_0(i),
        z1      => p21_1(i),
        z0      => p21_0(i),
        cy1     => cy11_1(i+1),
        cy0     => cy11_0(i+1)
                );
end generate leve1_1 ;

        op13_1 <= "00000000" & p13_1;
        op13_0 <= "00000000" & p13_0;
        op14_1 <= p14_1 & "00000000";
        op14_0 <= p14_0 & "00000000";

lv12_0_inst     : RBA_cell_PN
 port map(
        x1      => op13_1(0),
        x0      => op13_0(0),
        y1      => op14_1(0),
        y0      => op14_0(0),
        cin1    => zero,
        cin0    => zero,
        z1      => p22_1(0),
        z0      => p22_0(0),
        cy1     => cy12_1(1),
        cy0     => cy12_0(1)
                );

leve1_2: for i in 1 to 47 generate
lv12_i_inst     : RBA_cell_PN
 port map(
        x1      => op13_1(i),
        x0      => op13_0(i),
        y1      => op14_1(i),
        y0      => op14_0(i),
        cin1    => cy12_1(i),
        cin0    => cy12_0(i),
        z1      => p22_1(i),
        z0      => p22_0(i),
        cy1     => cy12_1(i+1),
        cy0     => cy12_0(i+1)
                );
end generate leve1_2 ;


                --level 2
        op21_1 <= "0000000000000000" &
p21_1;
        op21_0 <= "0000000000000000" &
p21_0;
        op22_1 <= p22_1 &
"0000000000000000";
        op22_0 <= p22_0 &
"0000000000000000";

lv2_0_inst      : RBA_cell_PN
 port map(
        x1      => op21_1(0),
        x0      => op21_0(0),
        y1      => op22_1(0),
        y0      => op22_0(0),
        cin1    => zero,
```

```
        cin0    => zero,
        z1      => p31_1(0),
        z0      => p31_0(0),
        cy1     => cy21_1(1),
        cy0     => cy21_0(1)
                );

leve2_1: for i in 1 to 63 generate
lv21_i_inst     : RBA_cell_PN
 port map(
        x1      => op21_1(i),
        x0      => op21_0(i),
        y1      => op22_1(i),
        y0      => op22_0(i),
        cin1    => cy21_1(i),
        cin0    => cy21_0(i),
        z1      => p31_1(i),
        z0      => p31_0(i),
        cy1     => cy21_1(i+1),
        cy0     => cy21_0(i+1)
                );
end generate leve2_1 ;



        rba_1 <= p31_1;
        rba_0 <= p31_0;


end rtl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.math_real.all;

entity RBA_cell_PN is
port(
        x1      : in  std_ulogic;
        x0      : in  std_ulogic;
        y1      : in  std_ulogic;
        y0      : in  std_ulogic;
        cin1    : in  std_ulogic;
        cin0    : in  std_ulogic;
        z1      : out std_ulogic;
        z0      : out std_ulogic;
        cy1     : out std_ulogic;
        cy0     : out std_ulogic
        );
end RBA_cell_PN;

architecture rtl of RBA_cell_PN is


        signal sx       : std_ulogic;
        signal sy       : std_ulogic;
        signal s        : std_ulogic;

begin

        sx      <= x1 xor x0;
        sy      <= y1 xor y0;
        s               <= sx xor sy;


        cy0     <= (s and not cin1) or (not s and
((X0 and Y0) or (not X1 and not Y1)));
        cy1     <= (not X0 and not Y0) or (X1
and Y1);
        z1      <= not (cin0 or not (s xor cin1));
        z0      <= not (not cin0 or (s xor cin1));

end rtl;
```

ANEXA 2

Codul sursă pentru blocul de înmulțire ce utilizează aritmetica normală

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity radix4 is
        port(

                signal multiplicand : in
std_logic_vector(7 downto 0);
                signal multiplicator : in
std_logic_vector(7 downto 0);
                signal rezultat: out
std_logic_vector(15 downto 0)

        );

end radix4;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;


architecture rtl of radix4 is

signal pp1_int: std_logic_vector(8 downto 0);
signal pp2_int: std_logic_vector(8 downto 0);
signal pp3_int: std_logic_vector(8 downto 0);
signal pp4_int: std_logic_vector(8 downto 0);

signal carry1_int: std_logic_vector(1 downto 0);
signal carry2_int: std_logic_vector(1 downto 0);
signal carry3_int: std_logic_vector(1 downto 0);
signal carry4_int: std_logic_vector(1 downto 0);

signal ppg1_int: std_logic_vector(15 downto 0);
signal ppg2_int: std_logic_vector(15 downto 0);
signal ppg3_int: std_logic_vector(15 downto 0);
signal ppg4_int: std_logic_vector(15 downto 0);

signal carry16_int: std_logic_vector(15 downto
0);

component  top_booth
        port(
multiplicand: in std_logic_vector(7 downto 0);
multiplicator: in std_logic_vector(7 downto 0);
pp1: out std_logic_vector (8 downto 0);
pp2: out std_logic_vector (8 downto 0);
pp3: out std_logic_vector (8 downto 0);
pp4: out std_logic_vector (8 downto 0);
carry1: out std_logic_vector(1 downto 0);
carry2: out std_logic_vector(1 downto 0);
carry3: out std_logic_vector(1 downto 0);
carry4: out std_logic_vector(1 downto 0)

        );

end component;

component pp_gen

        port(     pp1: in std_logic_vector (8
downto 0);
pp2: in std_logic_vector (8 downto 0);
pp3: in std_logic_vector (8 downto 0);
pp4: in std_logic_vector (8 downto 0);
carry1: in std_logic_vector(1 downto 0);
carry2: in std_logic_vector(1 downto 0);
carry3: in std_logic_vector(1 downto 0);
carry4: in std_logic_vector(1 downto 0);
ppg1: out std_logic_vector (15 downto 0);
ppg2: out std_logic_vector (15 downto 0);
ppg3: out std_logic_vector (15 downto 0);
ppg4: out std_logic_vector (15 downto 0);
carry16: out std_logic_vector (15 downto 0)

                );

end component;

component top_sum

        port(
ppg1: in std_logic_vector (15 downto 0);
ppg2: in std_logic_vector (15 downto 0);
ppg3: in std_logic_vector (15 downto 0);
ppg4: in std_logic_vector (15 downto 0);
carry_c: in std_logic_vector (15 downto 0);
```

```
final_c: out std_logic_vector (15 downto 0)

        );


end component;

begin

        U_TOP_BOOTH: top_booth port map(

        multiplicand    => multiplicand,
        multiplicator   => multiplicator,
        pp1             => pp1_int,
        pp2             => pp2_int,
        pp3             => pp3_int,
        pp4             => pp4_int,
        carry1          => carry1_int,
        carry2          => carry2_int,
        carry3          => carry3_int,
        carry4          => carry4_int

        );

        U_PP_GEN: pp_gen port map(
        pp1             => pp1_int,
        pp2             => pp2_int,
        pp3             => pp3_int,
        pp4             => pp4_int,
        carry1          => carry1_int,
        carry2          => carry2_int,
        carry3          => carry3_int,
        carry4          => carry4_int,
        ppg1            => ppg1_int,
        ppg2            => ppg2_int,
        ppg3            => ppg3_int,
        ppg4            => ppg4_int,
        carry16         => carry16_int


        );

        U_TOP_SUM: top_sum port map(
        ppg1            => ppg1_int,
        ppg2            => ppg2_int,
        ppg3            => ppg3_int,
        ppg4            => ppg4_int,
        carry_c         => carry16_int,
        final_c  => rezultat

        );

end rtl;

configuration radix4_cfg of radix4 is

        for rtl
        end for;

end radix4_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity top_booth is


port (
        multiplicand: in std_logic_vector(7
downto 0);
        multiplicator: in std_logic_vector(7
downto 0);
        pp1: out std_logic_vector (8 downto 0);
        pp2: out std_logic_vector (8 downto 0);
        pp3: out std_logic_vector (8 downto 0);
        pp4: out std_logic_vector (8 downto 0);
        carry1: out std_logic_vector(1 downto 0);
        carry2: out std_logic_vector(1 downto 0);
        carry3: out std_logic_vector(1 downto 0);
        carry4: out std_logic_vector(1 downto 0)

        );

end top_booth;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

architecture rtl of top_booth is

component booth
```

```
        port (
signal multiplicand : in std_logic_vector(7 downto
0);
signal yn_plus: in std_logic;
signal yn: in std_logic;
signal yn_minus: in std_logic;
signal pp: out std_logic_vector(8 downto 0);
signal carry: out std_logic_vector (1 downto 0)
        );

end component;

signal multiplicator9: std_logic_vector(8 downto
0);

begin

multiplicator9<= multiplicator &'0';

        B1: booth port map (
        multiplicand    => multiplicand,
        yn_plus         => multiplicator9(2),
        yn              => multiplicator9(1),
        yn_minus        => multiplicator9(0),
        pp              => pp1,
        carry           => carry1

        );

        B2: booth port map (
        multiplicand    => multiplicand,
        yn_plus         => multiplicator9(4),
        yn              => multiplicator9(3),
        yn_minus        => multiplicator9(2),
        pp              => pp2,
        carry           => carry2

        );


        B3: booth port map (

        multiplicand    => multiplicand,
        yn_plus         => multiplicator9(6),
        yn              => multiplicator9(5),
        yn_minus        => multiplicator9(4),
        pp              => pp3,
        carry           => carry3
        );

        B4: booth port map (

        multiplicand    => multiplicand,
        yn_plus         => multiplicator9(8),
        yn              => multiplicator9(7),
        yn_minus        => multiplicator9(6),
        pp              => pp4,
        carry           => carry4
        );


end rtl;

configuration top_booth_cfg of top_booth is

        for rtl
        end for;

end top_booth_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity booth is

        port(

        signal multiplicand : in std_logic_vector(7
downto 0);
        signal yn_plus: in std_logic;
        signal yn: in std_logic;
        signal yn_minus: in std_logic;
        signal pp: out std_logic_vector(8 downto
0);
        signal carry: out std_logic_vector (1
downto 0)


        );

end booth;

library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

architecture rtl of booth is

signal one_top: std_logic;
signal two_top: std_logic;
signal inv_top: std_logic;


component y_gen

        port(   signal yn_plus: in std_logic;
        signal yn: in std_logic;
        signal yn_minus: in std_logic;
        signal one: out std_logic;
        signal two: out std_logic;
        signal inv: out std_logic


        );

end component;

component  pp_9b

        port(   signal multiplicand : in
std_logic_vector(7 downto 0);
        signal one: in std_logic;
        signal two: in std_logic;
        signal inv: in std_logic;
        signal pp: out std_logic_vector(8 downto
0);
        signal carry: out std_logic_vector (1
downto 0)

        );

end component;

begin

        U_Y_GEN: y_gen port map(    yn_plus
=> yn_plus,
yn      => yn,
yn_minus => yn_minus,
one     => one_top,
two     => two_top,
inv     => inv_top


        );

U_PP_9B: pp_9b port map(
multiplicand => multiplicand,
one             => one_top,
two             => two_top,
inv             => inv_top,
pp              => pp,
carry           => carry


        );

end rtl;

configuration booth_cfg of booth is

        for rtl
        end for;

end booth_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;



entity y_gen is

        port(

signal yn_plus: in std_logic;
signal yn: in std_logic;
signal yn_minus: in std_logic;
signal one: out std_logic;
signal two: out std_logic;
signal inv: out std_logic

        );

end y_gen;

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

architecture rtl of y_gen is

signal one_int: std_logic;
signal two_int: std_logic;
signal inv_int: std_logic;

signal ynt_int:std_logic_vector(2 downto 0);
signal yn_plus_int: std_logic;
signal yn_int: std_logic;
signal yn_minus_int: std_logic;

begin

yn_plus_int<=yn_plus;
yn_int<=yn;
yn_minus_int<=yn_minus;

ynt_int<= yn_plus_int & yn_int & yn_minus_int;

pp_process:
process(ynt_int,one_int,two_int,inv_int)

begin

one_int<='0';
two_int<='0';
inv_int<='0';

case ynt_int is

        when "000" =>       one_int<='0';
                            two_int<='0';
                            inv_int<='0';

        when "001" =>       one_int<='1';
                            two_int<='0';
                            inv_int<='0';

        when "010" =>       one_int<='1';
                            two_int<='0';
                            inv_int<='0';

        when "011" =>       one_int<='0';
                            two_int<='1';
                            inv_int<='0';

        when "100" =>       one_int<='0';
                            two_int<='1';
                            inv_int<='1';

        when "101" =>       one_int<='1';
                            two_int<='0';
                            inv_int<='1';

        when "110" =>       one_int<='1';
                            two_int<='0';
                            inv_int<='1';

        when "111" =>       one_int<='0';
                            two_int<='0';
                            inv_int<='0';

        when others =>
                            one_int<='0';
                            two_int<='0';
                            inv_int<='0';

end case;


end process;

one<=one_int;
two<=two_int;
inv<=inv_int;

end rtl;

configuration y_gen_cfg of y_gen is

        for rtl
        end for;

end y_gen_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity pp_9b is
```

```
        port(

signal multiplicand : in std_logic_vector(7 downto
0);
signal one: in std_logic;
signal two: in std_logic;
signal inv: in std_logic;
signal pp: out std_logic_vector(8 downto 0);
signal carry: out std_logic_vector (1 downto 0)

        );

end pp_9b;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;


architecture rtl of pp_9b is

signal pp_int: std_logic_vector( 8 downto 0);
signal mult: std_logic_vector(1 downto 0);
signal carry_int: std_logic_vector(1 downto 0);

signal one_int: std_logic;
signal two_int: std_logic;
signal inv_int: std_logic;

signal pp_int_out: std_logic_vector( 8 downto 0);

begin


one_int<=one;
two_int<=two;
inv_int<=inv;

mult<= one_int & two_int;

pp_process: process (mult, multiplicand)

begin

case mult is
        when "10" =>
                pp_int<=multiplicand(7)
& multiplicand;


        when "01" =>
                pp_int<=multiplicand &
'0';


        when others =>
                pp_int<="000000000";

end case;

end process;

inv_process: process(pp_int, inv_int)

begin

if inv_int='1' then
        pp_int_out<=not(pp_int);

else pp_int_out<= pp_int;

end if;

end process;

carry_process: process(mult, inv_int)

begin

carry_int<="00";

 if inv_int='1' then
                case mult is

        when "10" =>
                carry_int<="01";

        when "01" =>
                carry_int<="10";

        when others =>
                carry_int<="00";


                end case;
```

```
end if;
end process;

pp<=pp_int_out;
carry<=carry_int;

end rtl;

configuration pp_9b_cfg of pp_9b is

        for rtl
        end for;

end pp_9b_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity pp_gen is

port (
pp1: in std_logic_vector (8 downto 0);
pp2: in std_logic_vector (8 downto 0);
pp3: in std_logic_vector (8 downto 0);
pp4: in std_logic_vector (8 downto 0);
carry1: in std_logic_vector(1 downto 0);
carry2: in std_logic_vector(1 downto 0);
carry3: in std_logic_vector(1 downto 0);
carry4: in std_logic_vector(1 downto 0);
ppg1: out std_logic_vector (15 downto 0);
ppg2: out std_logic_vector (15 downto 0);
ppg3: out std_logic_vector (15 downto 0);
ppg4: out std_logic_vector (15 downto 0);
carry16: out std_logic_vector (15 downto 0)
        );

end pp_gen;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

architecture rtl of pp_gen is

signal ppg1_int: std_logic_vector(15 downto 0);
signal ppg2_int: std_logic_vector(15 downto 0);
signal ppg3_int: std_logic_vector(15 downto 0);
signal ppg4_int: std_logic_vector(15 downto 0);
signal carry16_int: std_logic_vector (15 downto
0);

begin

ppg1_int<= pp1(8) & pp1(8) & pp1(8) & pp1(8)
& pp1(8) & pp1(8) & pp1(8) & pp1;

ppg2_int<= pp2(8) & pp2(8) & pp2(8) & pp2(8)
& pp2(8) & pp2 & "00";

ppg3_int<= pp3(8) & pp3(8) & pp3(8) & pp3 &
"0000";

ppg4_int<= pp4(8) & pp4 & "000000";

carry16_int<= "00000000" & carry4 & carry3 &
carry2 & carry1;

ppg1<=ppg1_int;
ppg2<=ppg2_int;
ppg3<=ppg3_int;
ppg4<=ppg4_int;
carry16<=carry16_int;

end rtl;

configuration pp_gen_cfg of pp_gen is

        for rtl
        end for;

end pp_gen_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
entity top_sum is
port (
ppg1: in std_logic_vector (15 downto 0);
ppg2: in std_logic_vector (15 downto 0);
ppg3: in std_logic_vector (15 downto 0);
```

```
ppg4: in std_logic_vector (15 downto 0);
carry_c: in std_logic_vector (15 downto 0);
final_c: out std_logic_vector (15 downto 0)
        );
end top_sum;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
architecture rtl of top_sum is
component sum

        port (
ppg_op1: in std_logic_vector (15 downto 0);
ppg_op2: in std_logic_vector (15 downto 0);
rez: out std_logic_vector (15 downto 0)
);
end component;
signal rez1: std_logic_vector(15 downto 0);
signal rez2: std_logic_vector(15 downto 0);
signal final: std_logic_vector(15 downto 0);
begin
S1: sum port map (
                ppg_op1 => ppg1,
                ppg_op2 => ppg2,
                rez => rez1
        );
S2: sum port map (
                ppg_op1 => ppg3,
                ppg_op2 => ppg4,
                rez => rez2

        );

        SF: sum port map (

                ppg_op1 => rez1,
                ppg_op2 => rez2,
                rez => final

        );

        SFC: sum port map (

                ppg_op1 => final,
                ppg_op2 => carry_c,
                rez => final_c
        );
end rtl;


configuration top_sum_cfg of top_sum is

        for rtl
        end for;

end top_sum_cfg;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

entity sum is
port (
        ppg_op1: in std_logic_vector (15 downto
0);
        ppg_op2: in std_logic_vector (15 downto
0);
        rez: out std_logic_vector (15 downto 0)
        );

end sum;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;

architecture rtl of sum is

signal sum_int: std_logic_vector(15 downto 0);
begin
sum_int<= ppg_op1 + ppg_op2;
rez<=sum_int;
end rtl;

configuration sum_cfg of sum is
        for rtl
        end for;

end sum_cfg;
```